BS ISO/IEC 14882:2024

BSI Standards Publication

Programming languages — C++

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2024

National foreword

This British Standard is the UK implementation of ISO/IEC 14882:2024.

The UK participation in its preparation was entrusted to Technical Committee IST/5, Programming languages, their environments and system software interfaces.

A list of organizations represented on this committee can be obtained on request to its committee manager.

Contractual and legal considerations

This publication has been prepared in good faith, however no representation, warranty, assurance or undertaking (express or implied) is or will be made, and no responsibility or liability is or will be accepted by BSI in relation to the adequacy, accuracy, completeness or reasonableness of this publication. All and any such responsibility and liability is expressly disclaimed to the full extent permitted by the law.

This publication is provided as is, and is to be used at the recipient's own risk.

The recipient is advised to consider seeking professional guidance with respect to its use of this publication.

This publication is not intended to constitute a contract. Users are responsible for its correct application.

© The British Standards Institution 2024
Published by BSI Standards Limited 2024

ISBN 978 0 539 19085 4

ICS 35.060

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 October 2024.

Amendments/corrigenda issued since publication

Date	Text affected

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2024

International Standard

ISO/IEC 14882

Programming languages — C++

Langages de programmation — C++

Seventh edition
2024-10

Reference number
ISO/IEC 14882:2024(en)

© ISO/IEC 2024

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2024

COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2024

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2024

Contents

Foreword xi

Introduction xiii

1 Scope 1

2 Normative references 2

3 Terms and definitions 3

4 General principles 9
4.1 Implementation compliance . . . 9
4.2 Structure of this document . . . 10
4.3 Syntax notation . . . 11

5 Lexical conventions 12
5.1 Separate translation . . . 12
5.2 Phases of translation . . . 12
5.3 Character sets . . . 13
5.4 Preprocessing tokens . . . 16
5.5 Alternative tokens . . . 17
5.6 Tokens . . . 17
5.7 Comments . . . 17
5.8 Header names . . . 17
5.9 Preprocessing numbers . . . 18
5.10 Identifiers . . . 18
5.11 Keywords . . . 19
5.12 Operators and punctuators . . . 20
5.13 Literals . . . 20

6 Basics 30
6.1 Preamble . . . 30
6.2 Declarations and definitions . . . 31
6.3 One-definition rule . . . 32
6.4 Scope . . . 37
6.5 Name lookup . . . 42
6.6 Program and linkage . . . 54
6.7 Memory and objects . . . 58
6.8 Types . . . 71
6.9 Program execution . . . 78

7 Expressions 90
7.1 Preamble . . . 90
7.2 Properties of expressions . . . 90
7.3 Standard conversions . . . 93
7.4 Usual arithmetic conversions . . . 98
7.5 Primary expressions . . . 99
7.6 Compound expressions . . . 116
7.7 Constant expressions . . . 148

8 Statements 157
8.1 Preamble . . . 157
8.2 Label . . . 158
8.3 Expression statement . . . 158

Documents
Document Sharing Hub

©ISO/IEC 2024 — All rights reserved

Full document details: BS ISO/IEC 14882:2024

8.4 Compound statement or block . . . 158
8.5 Selection statements . . . 158
8.6 Iteration statements . . . 161
8.7 Jump statements . . . 163
8.8 Declaration statement . . . 164
8.9 Ambiguity resolution . . . 165

9 Declarations **167**
9.1 Preamble . . . 167
9.2 Specifiers . . . 169
9.3 Declarators . . . 185
9.4 Initializers . . . 201
9.5 Function definitions . . . 217
9.6 Structured binding declarations . . . 222
9.7 Enumerations . . . 223
9.8 Namespaces . . . 227
9.9 The `using` declaration . . . 231
9.10 The `asm` declaration . . . 235
9.11 Linkage specifications . . . 236
9.12 Attributes . . . 238

10 Modules **246**
10.1 Module units and purviews . . . 246
10.2 Export declaration . . . 247
10.3 Import declaration . . . 250
10.4 Global module fragment . . . 251
10.5 Private module fragment . . . 253
10.6 Instantiation context . . . 254
10.7 Reachability . . . 255

11 Classes **257**
11.1 Preamble . . . 257
11.2 Properties of classes . . . 258
11.3 Class names . . . 259
11.4 Class members . . . 261
11.5 Unions . . . 282
11.6 Local class declarations . . . 284
11.7 Derived classes . . . 285
11.8 Member access control . . . 292
11.9 Initialization . . . 301
11.10 Comparisons . . . 313

12 Overloading **316**
12.1 Preamble . . . 316
12.2 Overload resolution . . . 316
12.3 Address of an overload set . . . 341
12.4 Overloaded operators . . . 343
12.5 Built-in operators . . . 346
12.6 User-defined literals . . . 348

13 Templates **350**
13.1 Preamble . . . 350
13.2 Template parameters . . . 351
13.3 Names of template specializations . . . 355
13.4 Template arguments . . . 357
13.5 Template constraints . . . 362
13.6 Type equivalence . . . 367
13.7 Template declarations . . . 368
13.8 Name resolution . . . 388

Documents
Document Sharing Hub

©ISO/IEC 2024 — All rights reserved

Full document details: BS ISO/IEC 14882:2024

13.9 Template instantiation and specialization . . . 401
13.10 Function template specializations . . . 413

14 Exception handling **433**
14.1 Preamble . . . 433
14.2 Throwing an exception . . . 434
14.3 Constructors and destructors . . . 435
14.4 Handling an exception . . . 436
14.5 Exception specifications . . . 438
14.6 Special functions . . . 440

15 Preprocessing directives **442**
15.1 Preamble . . . 442
15.2 Conditional inclusion . . . 444
15.3 Source file inclusion . . . 446
15.4 Module directive . . . 447
15.5 Header unit importation . . . 447
15.6 Macro replacement . . . 449
15.7 Line control . . . 454
15.8 Diagnostic directives . . . 454
15.9 Pragma directive . . . 455
15.10 Null directive . . . 455
15.11 Predefined macro names . . . 455
15.12 Pragma operator . . . 458

16 Library introduction **459**
16.1 General . . . 459
16.2 The C standard library . . . 460
16.3 Method of description . . . 460
16.4 Library-wide requirements . . . 467

17 Language support library **489**
17.1 General . . . 489
17.2 Common definitions . . . 489
17.3 Implementation properties . . . 493
17.4 Arithmetic types . . . 504
17.5 Startup and termination . . . 506
17.6 Dynamic memory management . . . 507
17.7 Type identification . . . 514
17.8 Source location . . . 515
17.9 Exception handling . . . 517
17.10 Initializer lists . . . 521
17.11 Comparisons . . . 522
17.12 Coroutines . . . 530
17.13 Other runtime support . . . 535
17.14 C headers . . . 537

18 Concepts library **539**
18.1 General . . . 539
18.2 Equality preservation . . . 539
18.3 Header `<concepts>` synopsis . . . 540
18.4 Language-related concepts . . . 542
18.5 Comparison concepts . . . 547
18.6 Object concepts . . . 550
18.7 Callable concepts . . . 550

19 Diagnostics library **552**
19.1 General . . . 552
19.2 Exception classes . . . 552

Documents
Document Sharing Hub

©ISO/IEC 2024 — All rights reserved

Full document details: BS ISO/IEC 14882:2024

19.3 Assertions . . . 555
19.4 Error numbers . . . 555
19.5 System error support . . . 557
19.6 Stacktrace . . . 565

20 Memory management library **572**
20.1 General . . . 572
20.2 Memory . . . 572
20.3 Smart pointers . . . 590
20.4 Memory resources . . . 617
20.5 Class template `scoped_allocator_adaptor` . . . 626

21 Metaprogramming library **630**
21.1 General . . . 630
21.2 Compile-time integer sequences . . . 630
21.3 Metaprogramming and type traits . . . 630
21.4 Compile-time rational arithmetic . . . 657

22 General utilities library **660**
22.1 General . . . 660
22.2 Utility components . . . 660
22.3 Pairs . . . 666
22.4 Tuples . . . 672
22.5 Optional objects . . . 686
22.6 Variants . . . 699
22.7 Storage for any type . . . 710
22.8 Expected objects . . . 715
22.9 Bitsets . . . 736
22.10 Function objects . . . 742
22.11 Class `type_index` . . . 769
22.12 Execution policies . . . 771
22.13 Primitive numeric conversions . . . 772
22.14 Formatting . . . 775
22.15 Bit manipulation . . . 801

23 Strings library **806**
23.1 General . . . 806
23.2 Character traits . . . 806
23.3 String view classes . . . 811
23.4 String classes . . . 821
23.5 Null-terminated sequence utilities . . . 849

24 Containers library **854**
24.1 General . . . 854
24.2 Requirements . . . 854
24.3 Sequence containers . . . 889
24.4 Associative containers . . . 920
24.5 Unordered associative containers . . . 939
24.6 Container adaptors . . . 963
24.7 Views . . . 1009

25 Iterators library **1036**
25.1 General . . . 1036
25.2 Header `<iterator>` synopsis . . . 1036
25.3 Iterator requirements . . . 1044
25.4 Iterator primitives . . . 1065
25.5 Iterator adaptors . . . 1068
25.6 Stream iterators . . . 1094
25.7 Range access . . . 1100

Documents
Document Sharing Hub

©ISO/IEC 2024 — All rights reserved

Full document details: BS ISO/IEC 14882:2024

26 Ranges library **1102**
26.1 General . . . 1102
26.2 Header `<ranges>` synopsis . . . 1102
26.3 Range access . . . 1111
26.4 Range requirements . . . 1115
26.5 Range utilities . . . 1118
26.6 Range factories . . . 1126
26.7 Range adaptors . . . 1139
26.8 Range generators . . . 1246

27 Algorithms library **1252**
27.1 General . . . 1252
27.2 Algorithms requirements . . . 1252
27.3 Parallel algorithms . . . 1254
27.4 Header `<algorithm>` synopsis . . . 1257
27.5 Algorithm result types . . . 1295
27.6 Non-modifying sequence operations . . . 1298
27.7 Mutating sequence operations . . . 1313
27.8 Sorting and related operations . . . 1330
27.9 Header `<numeric>` synopsis . . . 1356
27.10 Generalized numeric operations . . . 1360
27.11 Specialized `<memory>` algorithms . . . 1369
27.12 C library algorithms . . . 1375

28 Numerics library **1376**
28.1 General . . . 1376
28.2 Numeric type requirements . . . 1376
28.3 The floating-point environment . . . 1376
28.4 Complex numbers . . . 1377
28.5 Random number generation . . . 1384
28.6 Numeric arrays . . . 1426
28.7 Mathematical functions for floating-point types . . . 1445
28.8 Numbers . . . 1457

29 Time library **1459**
29.1 General . . . 1459
29.2 Header `<chrono>` synopsis . . . 1459
29.3 *Cpp17Clock* requirements . . . 1473
29.4 Time-related traits . . . 1474
29.5 Class template `duration` . . . 1475
29.6 Class template `time_point` . . . 1482
29.7 Clocks . . . 1485
29.8 The civil calendar . . . 1495
29.9 Class template `hh_mm_ss` . . . 1525
29.10 12/24 hours functions . . . 1527
29.11 Time zones . . . 1527
29.12 Formatting . . . 1540
29.13 Parsing . . . 1544
29.14 Header `<ctime>` synopsis . . . 1548

30 Localization library **1549**
30.1 General . . . 1549
30.2 Header `<locale>` synopsis . . . 1549
30.3 Locales . . . 1550
30.4 Standard `locale` categories . . . 1556
30.5 C library locales . . . 1588

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

31 Input/output library **1590**
31.1 General . . . 1590
31.2 Iostreams requirements . . . 1590
31.3 Forward declarations . . . 1591
31.4 Standard iostream objects . . . 1593
31.5 Iostreams base classes . . . 1595
31.6 Stream buffers . . . 1610
31.7 Formatting and manipulators . . . 1618
31.8 String-based streams . . . 1645
31.9 Span-based streams . . . 1659
31.10 File-based streams . . . 1666
31.11 Synchronized output streams . . . 1679
31.12 File systems . . . 1683
31.13 C library files . . . 1728

32 Regular expressions library **1732**
32.1 General . . . 1732
32.2 Requirements . . . 1732
32.3 Header `<regex>` synopsis . . . 1734
32.4 Namespace `std::regex_constants` . . . 1738
32.5 Class `regex_error` . . . 1741
32.6 Class template `regex_traits` . . . 1741
32.7 Class template `basic_regex` . . . 1744
32.8 Class template `sub_match` . . . 1747
32.9 Class template `match_results` . . . 1749
32.10 Regular expression algorithms . . . 1754
32.11 Regular expression iterators . . . 1758
32.12 Modified ECMAScript regular expression grammar . . . 1764

33 Concurrency support library **1766**
33.1 General . . . 1766
33.2 Requirements . . . 1766
33.3 Stop tokens . . . 1769
33.4 Threads . . . 1774
33.5 Atomic operations . . . 1782
33.6 Mutual exclusion . . . 1815
33.7 Condition variables . . . 1833
33.8 Semaphore . . . 1840
33.9 Coordination types . . . 1842
33.10 Futures . . . 1845

Annex A Grammar summary **1860**
A.1 General . . . 1860
A.2 Keywords . . . 1860
A.3 Lexical conventions . . . 1860
A.4 Basics . . . 1865
A.5 Expressions . . . 1865
A.6 Statements . . . 1869
A.7 Declarations . . . 1870
A.8 Modules . . . 1876
A.9 Classes . . . 1877
A.10 Overloading . . . 1878
A.11 Templates . . . 1878
A.12 Exception handling . . . 1879
A.13 Preprocessing directives . . . 1880

Annex B Implementation quantities **1882**

Documents
Document Sharing Hub

©ISO/IEC 2024 — All rights reserved

Full document details: BS ISO/IEC 14882:2024

Annex C Compatibility **1884**
C.1 C++ and ISO/IEC 14882:2020 . . . 1884
C.2 C++ and ISO/IEC 14882:2017 . . . 1888
C.3 C++ and ISO/IEC 14882:2014 . . . 1895
C.4 C++ and ISO/IEC 14882:2011 . . . 1899
C.5 C++ and ISO/IEC 14882:2003 . . . 1901
C.6 C++ and C . . . 1906
C.7 C standard library . . . 1914

Annex D Compatibility features **1917**
D.1 General . . . 1917
D.2 Arithmetic conversion on enumerations . . . 1917
D.3 Implicit capture of `*this` by reference . . . 1917
D.4 Array comparisons . . . 1917
D.5 Deprecated `volatile` types . . . 1917
D.6 Redeclaration of `static constexpr` data members . . . 1918
D.7 Non-local use of TU-local entities . . . 1918
D.8 Implicit declaration of copy functions . . . 1918
D.9 Literal operator function declarations using an identifier . . . 1919
D.10 `template` keyword before qualified names . . . 1919
D.11 Requires paragraph . . . 1919
D.12 `has_denorm` members in `numeric_limits` . . . 1919
D.13 Deprecated C macros . . . 1919
D.14 Relational operators . . . 1919
D.15 `char*` streams . . . 1920
D.16 Deprecated error numbers . . . 1927
D.17 The default allocator . . . 1928
D.18 Deprecated `polymorphic_allocator` member function . . . 1928
D.19 Deprecated type traits . . . 1928
D.20 Tuple . . . 1929
D.21 Variant . . . 1930
D.22 Deprecated `iterator` class template . . . 1930
D.23 Deprecated `move_iterator` access . . . 1930
D.24 Deprecated `shared_ptr` atomic access . . . 1931
D.25 Deprecated `basic_string` capacity . . . 1933
D.26 Deprecated standard code conversion facets . . . 1933
D.27 Deprecated convenience conversion interfaces . . . 1934
D.28 Deprecated locale category facets . . . 1938
D.29 Deprecated filesystem path factory functions . . . 1938
D.30 Deprecated atomic operations . . . 1938

Annex E Conformance with UAX #31 **1940**
E.1 General . . . 1940
E.2 R1 Default identifiers . . . 1940
E.3 R2 Immutable identifiers . . . 1940
E.4 R3 Pattern_White_Space and Pattern_Syntax characters . . . 1940
E.5 R4 Equivalent normalized identifiers . . . 1941
E.6 R5 Equivalent case-insensitive identifiers . . . 1941
E.7 R6 Filtered normalized identifiers . . . 1941
E.8 R7 Filtered case-insensitive identifiers . . . 1941
E.9 R8 Hashtag identifiers . . . 1941

Bibliography **1942**

Cross-references **1943**

Cross-references from ISO/IEC 14882:2020 **1970**

Index **1971**

Documents
Document Sharing Hub

©ISO/IEC 2024 — All rights reserved

Full document details: BS ISO/IEC 14882:2024

Index of grammar productions 2005

Index of library headers 2011

Index of library names 2013

Index of library concepts 2096

Index of implementation-defined behavior 2100

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see https://www.iso.org/directives or https://www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at https://www.iso.org/patents and https://patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see https://www.iso.org/iso/foreword.html. In the IEC, see https://www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

This seventh edition cancels and replaces the sixth edition (ISO/IEC 14882:2020), which has been technically revised.

The main changes are as follows:

— improved support for Unicode;

— improved support for programming with constant expressions and constant evaluation;

— addition of a new way to declare non-static member functions with an "explicit `this` parameter";

— addition of support for `#elifdef` and `#elifndef` preprocessing directives;

— change of overloaded `operator[]` to allow multiple parameters;

— change of lifetime rules in range-based for loops;

— addition of a new "decay-copying" declaration "`auto(x)`";

— support for extended floating-point types;

— addition of facilities for explicit lifetime management;

— addition of facilities for expressing assumptions;

— addition of standard library modules;

— addition of new standard library container and view types;

— addition of new standard library algorithms;

— addition of a generator type for use with coroutines;

— addition of an "expected" type for error handling;

— addition of string formatting and printing facilities;

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

— technical corrections and enhancements of existing core language and library facilities.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at https://www.iso.org/members.html and https://www.iec.ch/national-committees.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

Introduction

Clauses and subclauses in this document are annotated with a so-called stable name, presented in square brackets next to the (sub)clause heading (such as "[lex.token]" for 5.6, "Tokens"). Stable names aid in the discussion and evolution of this document by serving as stable references to subclauses across editions that are unaffected by changes of subclause numbering.

The cross references at the end of the document can be used to associate the stable names with their corresponding subclause number and to look up their location.

The indexes at the end of the document can be used to look up certain related entities such as grammar productions, names used by the standard library, and language constructs with implementation-defined behavior.

Aspects of the language syntax of C++ are distinguished typographically by the use of *italic*, *sans-serif* type or `constant width` type to avoid ambiguities; see 4.3.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

Documents
Document Sharing Hub

Programming languages — C++

1 Scope [intro.scope]

This document specifies requirements for implementations of the C++ programming language. The first such requirement is that an implementation implements the language, so this document also defines C++. Other requirements and relaxations of the first requirement appear at various places within this document.

C++ is a general purpose programming language based on ISO/IEC 9899:2018. C++ provides many facilities beyond those provided by ISO/IEC 9899:2018, including additional data types, classes, templates, exceptions, namespaces, operator overloading, function name overloading, references, free store management operators, and additional library facilities.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

2 Normative references [intro.refs]

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

— ISO/IEC 2382, *Information technology — Vocabulary*

— ISO 8601-1:2019, *Date and time — Representations for information interchange — Part 1: Basic rules*

— ISO/IEC 9899:2018, *Information technology — Programming languages — C*

— ISO/IEC/IEEE 9945:2009, *Information Technology — Portable Operating System Interface (POSIX®)*[1] *Base Specifications, Issue 7*

— ISO/IEC/IEEE 9945:2009/Cor 1:2013, *Information Technology — Portable Operating System Interface (POSIX®) Base Specifications, Issue 7 — Technical Corrigendum 1*

— ISO/IEC/IEEE 9945:2009/Cor 2:2017, *Information Technology — Portable Operating System Interface (POSIX®) Base Specifications, Issue 7 — Technical Corrigendum 2*

— ISO 80000-2:2019, *Quantities and units — Part 2: Mathematics*

— Ecma International, *ECMAScript*[2] *Language Specification*, Standard Ecma-262, third edition, 1999.

1) POSIX® is a registered trademark of the Institute of Electrical and Electronic Engineers, Inc. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of this product.

2) ECMAScript® is a registered trademark of Ecma International. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of this product.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

3 Terms and definitions [intro.defs]

For the purposes of this document, the terms and definitions given in ISO/IEC 2382, ISO 80000-2:2019, and the following apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

3.1 **[defns.access]**

access

⟨execution-time action⟩ read or modify the value of an object

Note 1 to entry: Only glvalues of scalar type can be used to access objects. Reads of scalar objects are described in 7.3.2 and modifications of scalar objects are described in 7.6.19, 7.6.1.6, and 7.6.2.3. Attempts to read or modify an object of class type typically invoke a constructor (11.4.5) or assignment operator (11.4.6); such invocations do not themselves constitute accesses, although they may involve accesses of scalar subobjects.

3.2 **[defns.argument]**

argument

⟨function call expression⟩ expression in the comma-separated list bounded by the parentheses

3.3 **[defns.argument.macro]**

argument

⟨function-like macro⟩ sequence of preprocessing tokens in the comma-separated list bounded by the parentheses

3.4 **[defns.argument.throw]**

argument

⟨throw expression⟩ operand of `throw`

3.5 **[defns.argument.templ]**

argument

⟨template instantiation⟩ *constant-expression*, *type-id*, or *id-expression* in the comma-separated list bounded by the angle brackets

3.6 **[defns.block]**

block

⟨execution⟩ wait for some condition (other than for the implementation to execute the execution steps of the thread of execution) to be satisfied before continuing execution past the blocking operation

3.7 **[defns.block.stmt]**

block

⟨statement⟩ compound statement

3.8 **[defns.c.lib]**

C standard library

library described in ISO/IEC 9899:2018, Clause 7

Note 1 to entry: With the qualifications noted in Clause 17 through Clause 33 and in C.7, the C standard library is a subset of the C++ standard library.

3.9 **[defns.character]**

character

⟨library⟩ object which, when treated sequentially, can represent text

Note 1 to entry: The term does not mean only `char`, `char8_t`, `char16_t`, `char32_t`, and `wchar_t` objects (6.8.2), but any value that can be represented by a type that provides the definitions specified in Clause 23, Clause 30, Clause 31, or Clause 32.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

3.10 **[defns.character.container]**

character container type

⟨library⟩ class or a type used to represent a *character* (3.9)

Note 1 to entry: It is used for one of the template parameters of the string, iostream, and regular expression class templates.

3.11 **[defns.regex.collating.element]**

collating element

sequence of one or more *character*s (3.9) within the current locale that collate as if they were a single character

3.12 **[defns.component]**

component

⟨library⟩ group of library entities directly related as members, *parameter*s (3.36), or return types

Note 1 to entry: For example, the class template `basic_string` and the non-member function templates that operate on strings are referred to as the string component.

3.13 **[defns.cond.supp]**

conditionally-supported

program construct that an implementation is not required to support

Note 1 to entry: Each implementation documents all conditionally-supported constructs that it does not support.

3.14 **[defns.const.subexpr]**

constant subexpression

expression whose evaluation as subexpression of a *conditional-expression* `CE` would not prevent `CE` from being a core constant expression

3.15 **[defns.deadlock]**

deadlock

⟨library⟩ situation wherein one or more threads are unable to continue execution because each is *block*ed (3.6) waiting for one or more of the others to satisfy some condition

3.16 **[defns.default.behavior.impl]**

default behavior

⟨library implementation⟩ specific behavior provided by the implementation, within the scope of the *required behavior* (3.46)

3.17 **[defns.diagnostic]**

diagnostic message

message belonging to an implementation-defined subset of the implementation's output messages

3.18 **[defns.dynamic.type]**

dynamic type

⟨glvalue⟩ type of the most derived object to which the glvalue refers

EXAMPLE If a pointer (9.3.4.2) `p` whose static type is "pointer to class `B`" is pointing to an object of class `D`, derived from `B` (11.7), the dynamic type of the expression `*p` is "`D`". References (9.3.4.3) are treated similarly.

3.19 **[defns.dynamic.type.prvalue]**

dynamic type

⟨prvalue⟩ *static type* (3.58) of the prvalue expression

3.20 **[defns.expression.equivalent]**

expression-equivalent

⟨library⟩ expressions that all have the same effects, either are all potentially-throwing or are all not potentially-throwing, and either are all *constant subexpression*s (3.14) or are all not constant subexpressions

EXAMPLE For a value `x` of type `int` and a function `f` that accepts integer arguments, the expressions `f(x + 2)`, `f(2 + x)`, and `f(1 + x + 1)` are expression-equivalent.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

3.21 **[defns.regex.finite.state.machine]**
finite state machine
⟨regular expression⟩ unspecified data structure that is used to represent a *regular expression* (3.44), and which permits efficient matches against the regular expression to be obtained

3.22 **[defns.regex.format.specifier]**
format specifier
⟨regular expression⟩ sequence of one or more *character*s (3.9) that is expected to be replaced with some part of a *regular expression* (3.44) match

3.23 **[defns.handler]**
handler function
⟨library⟩ non-reserved function whose definition may be provided by a C++ program

Note 1 to entry: A C++ program may designate a handler function at various points in its execution by supplying a pointer to the function when calling any of the library functions that install handler functions (see Clause 17).

3.24 **[defns.ill.formed]**
ill-formed program
program that is not well-formed (3.65)

3.25 **[defns.impl.defined]**
implementation-defined behavior
behavior, for a *well-formed program* (3.65) construct and correct data, that depends on the implementation and that each implementation documents

3.26 **[defns.order.ptr]**
implementation-defined strict total order over pointers
⟨library⟩ implementation-defined strict total ordering over all pointer values such that the ordering is consistent with the partial order imposed by the built-in operators `<`, `>`, `<=`, `>=`, and `<=>`

3.27 **[defns.impl.limits]**
implementation limit
restriction imposed upon programs by the implementation

3.28 **[defns.locale.specific]**
locale-specific behavior
behavior that depends on local conventions of nationality, culture, and language that each implementation documents

3.29 **[defns.regex.matched]**
matched
⟨regular expression⟩ condition when a sequence of zero or more *character*s (3.9) correspond to a sequence of characters defined by the pattern

3.30 **[defns.modifier]**
modifier function
⟨library⟩ class member function other than a constructor, assignment operator, or destructor that alters the state of an object of the class

3.31 **[defns.move.assign]**
move assignment
⟨library⟩ assignment of an rvalue of some object type to a modifiable lvalue of the same type

3.32 **[defns.move.constr]**
move construction
⟨library⟩ direct-initialization of an object of some type with an rvalue of the same type

3.33 **[defns.nonconst.libcall]**
non-constant library call
invocation of a library function that, as part of evaluating any expression `E`, prevents `E` from being a core constant expression

Documents
Document Sharing Hub

©ISO/IEC 2024 — All rights reserved

Full document details: BS ISO/IEC 14882:2024

3.34 **[defns.ntcts]**
NTCTS
⟨library⟩ sequence of values that have *character* (3.9) type that precede the terminating null character type value `charT()`

3.35 **[defns.observer]**
observer function
⟨library⟩ class member function that accesses the state of an object of the class but does not alter that state

Note 1 to entry: Observer functions are specified as `const` member functions.

3.36 **[defns.parameter]**
parameter
⟨function or catch clause⟩ object or reference declared as part of a function declaration or definition or in the catch clause of an exception handler that acquires a value on entry to the function or handler

3.37 **[defns.parameter.macro]**
parameter
⟨function-like macro⟩ identifier from the comma-separated list bounded by the parentheses immediately following the macro name

3.38 **[defns.parameter.templ]**
parameter
⟨template⟩ member of a *template-parameter-list*

3.39 **[defns.regex.primary.equivalence.class]**
primary equivalence class
⟨regular expression⟩ set of one or more *character*s (3.9) which share the same primary sort key: that is the sort key weighting that depends only upon character shape, and not accents, case, or locale specific tailorings

3.40 **[defns.prog.def.spec]**
program-defined specialization
⟨library⟩ explicit template specialization or partial specialization that is not part of the C++ standard library and not defined by the implementation

3.41 **[defns.prog.def.type]**
program-defined type
⟨library⟩ non-closure class type or enumeration type that is not part of the C++ standard library and not defined by the implementation, or a closure type of a non-implementation-provided lambda expression, or an instantiation of a *program-defined specialization* (3.40)

Note 1 to entry: Types defined by the implementation include extensions (4.1) and internal types used by the library.

3.42 **[defns.projection]**
projection
⟨library⟩ transformation that an algorithm applies before inspecting the values of elements

EXAMPLE

```
std::pair<int, std::string_view> pairs[] = {{2, "foo"}, {1, "bar"}, {0, "baz"}};
std::ranges::sort(pairs, std::ranges::less{}, [](auto const& p) { return p.first; });
```

sorts the pairs in increasing order of their `first` members:

```
{{0, "baz"}, {1, "bar"}, {2, "foo"}}
```

3.43 **[defns.referenceable]**
referenceable type
type that is either an object type, a function type that does not have cv-qualifiers or a *ref-qualifier*, or a reference type

Note 1 to entry: The term describes a type to which a reference can be created, including reference types.

3.44 **[defns.regex.regular.expression]**
regular expression
pattern that selects specific strings from a set of *character* (3.9) strings

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

3.45 **[defns.replacement]**
replacement function
⟨library⟩ non-reserved function whose definition is provided by a C++ program

Note 1 to entry: Only one definition for such a function is in effect for the duration of the program's execution, as the result of creating the program (5.2) and resolving the definitions of all translation units (6.6).

3.46 **[defns.required.behavior]**
required behavior
⟨library⟩ description of *replacement function* (3.45) and *handler function* (3.23) semantics applicable to both the behavior provided by the implementation and the behavior of any such function definition in the program

Note 1 to entry: If such a function defined in a C++ program fails to meet the required behavior when it executes, the behavior is undefined.

3.47 **[defns.reserved.function]**
reserved function
⟨library⟩ function, specified as part of the C++ standard library, that is defined by the implementation

Note 1 to entry: If a C++ program provides a definition for any reserved function, the results are undefined.

3.48 **[defns.signature]**
signature
⟨function⟩ name, parameter-type-list, and enclosing namespace

Note 1 to entry: Signatures are used as a basis for name mangling and linking.

3.49 **[defns.signature.friend]**
signature
⟨non-template friend function with trailing *requires-clause*⟩ name, parameter-type-list, enclosing class, and trailing *requires-clause*

3.50 **[defns.signature.templ]**
signature
⟨function template⟩ name, parameter-type-list, enclosing namespace, return type, *signature* (3.56) of the *template-head*, and trailing *requires-clause* (if any)

3.51 **[defns.signature.templ.friend]**
signature
⟨friend function template with constraint involving enclosing template parameters⟩ name, parameter-type-list, return type, enclosing class, *signature* (3.56) of the *template-head*, and trailing *requires-clause* (if any)

3.52 **[defns.signature.spec]**
signature
⟨function template specialization⟩ *signature* (3.50) of the template of which it is a specialization and its template *arguments* (3.5) (whether explicitly specified or deduced)

3.53 **[defns.signature.member]**
signature
⟨class member function⟩ name, parameter-type-list, class of which the function is a member, *cv*-qualifiers (if any), *ref-qualifier* (if any), and trailing *requires-clause* (if any)

3.54 **[defns.signature.member.templ]**
signature
⟨class member function template⟩ name, parameter-type-list, class of which the function is a member, *cv*-qualifiers (if any), *ref-qualifier* (if any), return type (if any), *signature* (3.56) of the *template-head*, and trailing *requires-clause* (if any)

3.55 **[defns.signature.member.spec]**
signature
⟨class member function template specialization⟩ *signature* (3.54) of the member function template of which it is a specialization and its template arguments (whether explicitly specified or deduced)

Documents
Document Sharing Hub

©ISO/IEC 2024 — All rights reserved

Full document details: BS ISO/IEC 14882:2024

3.56 **[defns.signature.template.head]**
signature
⟨*template-head*⟩ template *parameter* (3.38) list, excluding template parameter names and default *arguments* (3.5), and *requires-clause* (if any)

3.57 **[defns.stable]**
stable algorithm
⟨library⟩ algorithm that preserves, as appropriate to the particular algorithm, the order of elements

Note 1 to entry: Requirements for stable algorithms are given in 16.4.6.8.

3.58 **[defns.static.type]**
static type
type of an expression resulting from analysis of the program without considering execution semantics

Note 1 to entry: The static type of an expression depends only on the form of the program in which the expression appears, and does not change while the program is executing.

3.59 **[defns.regex.subexpression]**
sub-expression
⟨regular expression⟩ subset of a *regular expression* (3.44) that has been marked by parentheses

3.60 **[defns.traits]**
traits class
⟨library⟩ class that encapsulates a set of types and functions necessary for class templates and function templates to manipulate objects of types for which they are instantiated

3.61 **[defns.unblock]**
unblock
satisfy a condition that one or more *blocked* (3.6) threads of execution are waiting for

3.62 **[defns.undefined]**
undefined behavior
behavior for which this document imposes no requirements

Note 1 to entry: Undefined behavior may be expected when this document omits any explicit definition of behavior or when a program uses an erroneous construct or erroneous data. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a *diagnostic message* (3.17)), to terminating a translation or execution (with the issuance of a diagnostic message). Many erroneous program constructs do not engender undefined behavior; they are required to be diagnosed. Evaluation of a constant expression (7.7) never exhibits behavior explicitly specified as undefined in Clause 4 through Clause 15.

3.63 **[defns.unspecified]**
unspecified behavior
behavior, for a *well-formed program* (3.65) construct and correct data, that depends on the implementation

Note 1 to entry: The implementation is not required to document which behavior occurs. The range of possible behaviors is usually delineated by this document.

3.64 **[defns.valid]**
valid but unspecified state
⟨library⟩ value of an object that is not specified except that the object's invariants are met and operations on the object behave as specified for its type

EXAMPLE If an object `x` of type `std::vector<int>` is in a valid but unspecified state, `x.empty()` can be called unconditionally, and `x.front()` can be called only if `x.empty()` returns `false`.

3.65 **[defns.well.formed]**
well-formed program
C++ program constructed according to the syntax and semantic rules

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

4 General principles [intro]

4.1 Implementation compliance [intro.compliance]

4.1.1 General [intro.compliance.general]

The set of *diagnosable rules* consists of all syntactic and semantic rules in this document except for those rules containing an explicit notation that "no diagnostic is required" or which are described as resulting in "undefined behavior".

Although this document states only requirements on C++ implementations, those requirements are often easier to understand if they are phrased as requirements on programs, parts of programs, or execution of programs. Such requirements have the following meaning:

— If a program contains no violations of the rules in Clause 5 through Clause 33 as well as those specified in Annex D, a conforming implementation shall, in accordance with the resource limits specified in Annex B, accept and correctly execute[3] that program.

— If a program contains a violation of a rule for which no diagnostic is required, this document places no requirement on implementations with respect to that program.

— Otherwise, if a program contains a violation of any diagnosable rule or an occurrence of a construct described in this document as "conditionally-supported" when the implementation does not support that construct, a conforming implementation shall issue at least one diagnostic message.

NOTE During template argument deduction and substitution, certain constructs that in other contexts require a diagnostic are treated differently; see 13.10.3.

Furthermore, a conforming implementation

— shall not accept a preprocessing translation unit containing a `#error` preprocessing directive (15.8),

— shall issue at least one diagnostic message for each `#warning` or `#error` preprocessing directive not following a `#error` preprocessing directive in a preprocessing translation unit, and

— shall not accept a translation unit with a *static_assert-declaration* that fails (9.1).

For classes and class templates, the library Clauses specify partial definitions. Private members (11.8) are not specified, but each implementation shall supply them to complete the definitions according to the description in the library Clauses.

For functions, function templates, objects, and values, the library Clauses specify declarations. Implementations shall supply definitions consistent with the descriptions in the library Clauses.

A C++ translation unit (5.2) obtains access to the names defined in the library by including the appropriate standard library header or importing the appropriate standard library named header unit (16.4.3.2).

The templates, classes, functions, and objects in the library have external linkage (6.6). The implementation provides definitions for standard library entities, as necessary, while combining translation units to form a complete C++ program (5.2).

Two kinds of implementations are defined: a *hosted implementation* and a *freestanding implementation*. A freestanding implementation is one in which execution may take place without the benefit of an operating system. A hosted implementation supports all the facilities described in this document, while a freestanding implementation supports the entire C++ language described in Clause 5 through Clause 15 and the subset of the library facilities described in 16.4.2.5.

A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any well-formed program. Implementations are required to diagnose programs that use such extensions that are ill-formed according to this document. Having done so, however, they can compile and execute such programs.

Each implementation shall include documentation that identifies all conditionally-supported constructs that it does not support and defines all locale-specific characteristics.[4]

3) "Correct execution" can include undefined behavior, depending on the data being processed; see Clause 3 and 6.9.1.
4) This documentation also defines implementation-defined behavior; see 4.1.2.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

4.1.2 Abstract machine [intro.abstract]

The semantic descriptions in this document define a parameterized nondeterministic abstract machine. This document places no requirement on the structure of conforming implementations. In particular, they need not copy or emulate the structure of the abstract machine. Rather, conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below.[5]

Certain aspects and operations of the abstract machine are described in this document as implementation-defined behavior (for example, `sizeof(int)`). These constitute the parameters of the abstract machine. Each implementation shall include documentation describing its characteristics and behavior in these respects.[6] Such documentation shall define the instance of the abstract machine that corresponds to that implementation (referred to as the "corresponding instance" below).

Certain other aspects and operations of the abstract machine are described in this document as unspecified behavior (for example, order of evaluation of arguments in a function call (7.6.1.3)). Where possible, this document defines a set of allowable behaviors. These define the nondeterministic aspects of the abstract machine. An instance of the abstract machine can thus have more than one possible execution for a given program and a given input.

Certain other operations are described in this document as undefined behavior (for example, the effect of attempting to modify a const object).

NOTE 1 This document imposes no requirements on the behavior of programs that contain undefined behavior.

A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution contains an undefined operation, this document places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).

The least requirements on a conforming implementation are:

— Accesses through volatile glvalues are evaluated strictly according to the rules of the abstract machine.

— At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.

— The input and output dynamics of interactive devices shall take place in such a fashion that prompting output is actually delivered before a program waits for input. What constitutes an interactive device is implementation-defined.

These collectively are referred to as the *observable behavior* of the program.

NOTE 2 More stringent correspondences between abstract and actual semantics can be defined by each implementation.

4.2 Structure of this document [intro.structure]

Clause 5 through Clause 15 describe the C++ programming language. That description includes detailed syntactic specifications in a form described in 4.3. For convenience, Annex A repeats all such syntactic specifications.

Clause 17 through Clause 33 and Annex D (the *library clauses*) describe the C++ standard library. That description includes detailed descriptions of the entities and macros that constitute the library, in a form described in Clause 16.

Annex B recommends lower bounds on the capacity of conforming implementations.

Annex C summarizes the evolution of C++ since its first published description, and explains in detail the differences between C++ and C. Certain features of C++ exist solely for compatibility purposes; Annex D describes those features.

5) This provision is sometimes called the "as-if" rule, because an implementation is free to disregard any requirement of this document as long as the result is *as if* the requirement had been obeyed, as far as can be determined from the observable behavior of the program. For instance, an actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no side effects affecting the observable behavior of the program are produced.

6) This documentation also includes conditionally-supported constructs and locale-specific behavior. See 4.1.1.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

4.3 Syntax notation [syntax]

In the syntax notation used in this document, syntactic categories are indicated by *italic, sans-serif* type, and literal words and characters in `constant width` type. Alternatives are listed on separate lines except in a few cases where a long set of alternatives is marked by the phrase "one of". If the text of an alternative is too long to fit on a line, the text is continued on subsequent lines indented from the first one. An optional terminal or non-terminal symbol is indicated by the subscript "$_{opt}$", so

`{` *expression*$_{opt}$ `}`

indicates an optional expression enclosed in braces.

Names for syntactic categories have generally been chosen according to the following rules:

— *X-name* is a use of an identifier in a context that determines its meaning (e.g., *class-name*, *typedef-name*).

— *X-id* is an identifier with no context-dependent meaning (e.g., *qualified-id*).

— *X-seq* is one or more *X*'s without intervening delimiters (e.g., *declaration-seq* is a sequence of declarations).

— *X-list* is one or more *X*'s separated by intervening commas (e.g., *identifier-list* is a sequence of identifiers separated by commas).

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

5 Lexical conventions [lex]

5.1 Separate translation [lex.separate]

The text of the program is kept in units called *source files* in this document. A source file together with all the headers (16.4.2.3) and source files included (15.3) via the preprocessing directive `#include`, less any source lines skipped by any of the conditional inclusion (15.2) preprocessing directives, is called a *preprocessing translation unit*.

NOTE 1 A C++ program need not all be translated at the same time.

NOTE 2 Previously translated translation units and instantiation units can be preserved individually or in libraries. The separate translation units of a program communicate (6.6) by (for example) calls to functions whose identifiers have external or module linkage, manipulation of objects whose identifiers have external or module linkage, or manipulation of data files. Translation units can be separately translated and then later linked to produce an executable program (6.6).

5.2 Phases of translation [lex.phases]

The precedence among the syntax rules of translation is specified by the following phases.[7]

1. An implementation shall support input files that are a sequence of UTF-8 code units (UTF-8 files). It may also support an implementation-defined set of other kinds of input files, and, if so, the kind of an input file is determined in an implementation-defined manner that includes a means of designating input files as UTF-8 files, independent of their content.

   NOTE 1 In other words, recognizing the U+FEFF BYTE ORDER MARK is not sufficient.

   If an input file is determined to be a UTF-8 file, then it shall be a well-formed UTF-8 code unit sequence and it is decoded to produce a sequence of Unicode[8] scalar values. A sequence of translation character set elements is then formed by mapping each Unicode scalar value to the corresponding translation character set element. In the resulting sequence, each pair of characters in the input sequence consisting of U+000D CARRIAGE RETURN followed by U+000A LINE FEED, as well as each U+000D CARRIAGE RETURN not immediately followed by a U+000A LINE FEED, is replaced by a single new-line character.

   For any other kind of input file supported by the implementation, characters are mapped, in an implementation-defined manner, to a sequence of translation character set elements (5.3), representing end-of-line indicators as new-line characters.

2. If the first translation character is U+FEFF BYTE ORDER MARK, it is deleted. Each sequence of a backslash character (\) immediately followed by zero or more whitespace characters other than new-line followed by a new-line character is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice. Except for splices reverted in a raw string literal, if a splice results in a character sequence that matches the syntax of a *universal-character-name*, the behavior is undefined. A source file that is not empty and that does not end in a new-line character, or that ends in a splice, shall be processed as if an additional new-line character were appended to the file.

3. The source file is decomposed into preprocessing tokens (5.4) and sequences of whitespace characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment.[9] Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of whitespace characters other than new-line is retained or replaced by one space character is unspecified. As characters from the source file are consumed to form the next preprocessing token (i.e., not being consumed as part of a comment or other forms of whitespace), except when matching a *c-char-sequence*, *s-char-sequence*, *r-char-sequence*, *h-char-sequence*, or *q-char-sequence*, *universal-character-name*s are recognized and replaced by the designated element of the

7) Implementations behave as if these separate phases occur, although in practice different phases can be folded together.

8) Unicode® is a registered trademark of Unicode, Inc. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of this product.

9) A partial preprocessing token would arise from a source file ending in the first portion of a multi-character token that requires a terminating sequence of characters, such as a *header-name* that is missing the closing " or >. A partial comment would arise from a source file ending with an unclosed /* comment.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

translation character set. The process of dividing a source file's characters into preprocessing tokens is context-dependent.

EXAMPLE See the handling of `<` within a `#include` preprocessing directive.

4. Preprocessing directives are executed, macro invocations are expanded, and `_Pragma` unary operator expressions are executed. A `#include` preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively. All preprocessing directives are then deleted.
5. For a sequence of two or more adjacent *string-literal* tokens, a common *encoding-prefix* is determined as specified in 5.13.5. Each such *string-literal* token is then considered to have that common *encoding-prefix*.
6. Adjacent *string-literal* tokens are concatenated (5.13.5).
7. Whitespace characters separating tokens are no longer significant. Each preprocessing token is converted into a token (5.6). The resulting tokens constitute a *translation unit* and are syntactically and semantically analyzed and translated.

   NOTE 2 The process of analyzing and translating the tokens can occasionally result in one token being replaced by a sequence of other tokens (13.3).

   It is implementation-defined whether the sources for module units and header units on which the current translation unit has an interface dependency (10.1, 10.3) are required to be available.

   NOTE 3 Source files, translation units and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation.
8. Translated translation units and instantiation units are combined as follows:

   NOTE 4 Some or all of these can be supplied from a library.

   Each translated translation unit is examined to produce a list of required instantiations.

   NOTE 5 This can include instantiations which have been explicitly requested (13.9.3).

   The definitions of the required templates are located. It is implementation-defined whether the source of the translation units containing these definitions is required to be available.

   NOTE 6 An implementation can choose to encode sufficient information into the translated translation unit so as to ensure the source is not required here.

   All the required instantiations are performed to produce *instantiation units*.

   NOTE 7 These are similar to translated translation units, but contain no references to uninstantiated templates and no template definitions.

   The program is ill-formed if any instantiation fails.
9. All external entity references are resolved. Library components are linked to satisfy external references to entities not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.

5.3 Character sets [lex.charset]

The *translation character set* consists of the following elements:

— each abstract character assigned a code point in the Unicode codespace as specified in the Unicode Standard[14], and

— a distinct character for each Unicode scalar value not assigned to an abstract character.

NOTE 1 Unicode code points are integers in the range $[0, 10FFFF]$ (hexadecimal). A surrogate code point is a value in the range $[D800, DFFF]$ (hexadecimal). A Unicode scalar value is any code point that is not a surrogate code point.

The *basic character set* is a subset of the translation character set, consisting of 96 characters as specified in Table 1.

NOTE 2 Unicode short names are given only as a means to identifying the character; the numerical value has no other meaning in this context.

The *universal-character-name* construct provides a way to name other characters.

n-char: one of
  any member of the translation character set except the U+007D RIGHT CURLY BRACKET or new-line character

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

Table 1 — Basic character set [tab:lex.charset.basic]

character		glyph
U+0009	CHARACTER TABULATION	
U+000B	LINE TABULATION	
U+000C	FORM FEED	
U+0020	SPACE	
U+000A	LINE FEED	new-line
U+0021	EXCLAMATION MARK	!
U+0022	QUOTATION MARK	"
U+0023	NUMBER SIGN	#
U+0025	PERCENT SIGN	%
U+0026	AMPERSAND	&
U+0027	APOSTROPHE	'
U+0028	LEFT PARENTHESIS	(
U+0029	RIGHT PARENTHESIS	)
U+002A	ASTERISK	*
U+002B	PLUS SIGN	+
U+002C	COMMA	,
U+002D	HYPHEN-MINUS	-
U+002E	FULL STOP	.
U+002F	SOLIDUS	/
U+0030 .. U+0039	DIGIT ZERO .. NINE	0 1 2 3 4 5 6 7 8 9
U+003A	COLON	:
U+003B	SEMICOLON	;
U+003C	LESS-THAN SIGN	<
U+003D	EQUALS SIGN	=
U+003E	GREATER-THAN SIGN	>
U+003F	QUESTION MARK	?
U+0041 .. U+005A	LATIN CAPITAL LETTER A .. Z	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
U+005B	LEFT SQUARE BRACKET	[
U+005C	REVERSE SOLIDUS	\
U+005D	RIGHT SQUARE BRACKET	]
U+005E	CIRCUMFLEX ACCENT	^
U+005F	LOW LINE	_
U+0061 .. U+007A	LATIN SMALL LETTER A .. Z	a b c d e f g h i j k l m n o p q r s t u v w x y z
U+007B	LEFT CURLY BRACKET	{
U+007C	VERTICAL LINE	|
U+007D	RIGHT CURLY BRACKET	}
U+007E	TILDE	~

n-char-sequence:
  n-char
  n-char-sequence n-char

named-universal-character:
  `\N{` *n-char-sequence* `}`

hex-quad:
  hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

simple-hexadecimal-digit-sequence:
  hexadecimal-digit
  simple-hexadecimal-digit-sequence hexadecimal-digit

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

universal-character-name:
  \u *hex-quad*
  \U *hex-quad hex-quad*
  \u{ *simple-hexadecimal-digit-sequence* }
  named-universal-character

A *universal-character-name* of the form \u *hex-quad*, \U *hex-quad hex-quad*, or \u{*simple-hexadecimal-digit-sequence*} designates the character in the translation character set whose Unicode scalar value is the hexadecimal number represented by the sequence of *hexadecimal-digit*s in the *universal-character-name*. The program is ill-formed if that number is not a Unicode scalar value.

A *universal-character-name* that is a *named-universal-character* designates the corresponding character in the Unicode Standard (chapter 4.8 Name) if the *n-char-sequence* is equal to its character name or to one of its character name aliases of type "control", "correction", or "alternate"; otherwise, the program is ill-formed.

NOTE 3 These aliases are listed in the Unicode Character Database's `NameAliases.txt`. None of these names or aliases have leading or trailing spaces.

If a *universal-character-name* outside the *c-char-sequence*, *s-char-sequence*, or *r-char-sequence* of a *character-literal* or *string-literal* (in either case, including within a *user-defined-literal*) corresponds to a control character or to a character in the basic character set, the program is ill-formed.

NOTE 4 A sequence of characters resembling a *universal-character-name* in an *r-char-sequence* (5.13.5) does not form a *universal-character-name*.

The *basic literal character set* consists of all characters of the basic character set, plus the control characters specified in Table 2.

Table 2 — Additional control characters [tab:lex.charset.literal]

character	
U+0000	NULL
U+0007	ALERT
U+0008	BACKSPACE
U+000D	CARRIAGE RETURN

A *code unit* is an integer value of character type (6.8.2). Characters in a *character-literal* other than a multicharacter or non-encodable character literal or in a *string-literal* are encoded as a sequence of one or more code units, as determined by the *encoding-prefix* (5.13.3, 5.13.5); this is termed the respective *literal encoding*. The *ordinary literal encoding* is the encoding applied to an ordinary character or string literal. The *wide literal encoding* is the encoding applied to a wide character or string literal.

A literal encoding or a locale-specific encoding of one of the execution character sets (16.3.3.3.4) encodes each element of the basic literal character set as a single code unit with non-negative value, distinct from the code unit for any other such element.

NOTE 5 A character not in the basic literal character set can be encoded with more than one code unit; the value of such a code unit can be the same as that of a code unit for an element of the basic literal character set.

The U+0000 NULL character is encoded as the value `0`. No other element of the translation character set is encoded with a code unit of value `0`. The code unit value of each decimal digit character after the digit `0` (U+0030) shall be one greater than the value of the previous. The ordinary and wide literal encodings are otherwise implementation-defined. For a UTF-8, UTF-16, or UTF-32 literal, the implementation shall encode the Unicode scalar value corresponding to each character of the translation character set as specified in the Unicode Standard for the respective Unicode encoding form.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

5.4 Preprocessing tokens [lex.pptoken]

preprocessing-token:
- *header-name*
- *import-keyword*
- *module-keyword*
- *export-keyword*
- *identifier*
- *pp-number*
- *character-literal*
- *user-defined-character-literal*
- *string-literal*
- *user-defined-string-literal*
- *preprocessing-op-or-punc*
- each non-whitespace character that cannot be one of the above

Each preprocessing token that is converted to a token (5.6) shall have the lexical form of a keyword, an identifier, a literal, or an operator or punctuator.

A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. In this document, glyphs are used to identify elements of the basic character set (5.3). The categories of preprocessing token are: header names, placeholder tokens produced by preprocessing `import` and `module` directives (*import-keyword*, *module-keyword*, and *export-keyword*), identifiers, preprocessing numbers, character literals (including user-defined character literals), string literals (including user-defined string literals), preprocessing operators and punctuators, and single non-whitespace characters that do not lexically match the other preprocessing token categories. If a U+0027 APOSTROPHE or a U+0022 QUOTATION MARK character matches the last category, the behavior is undefined. If any character not in the basic character set matches the last category, the program is ill-formed. Preprocessing tokens can be separated by whitespace; this consists of comments (5.7), or whitespace characters (U+0020 SPACE, U+0009 CHARACTER TABULATION, new-line, U+000B LINE TABULATION, and U+000C FORM FEED), or both. As described in Clause 15, in certain circumstances during translation phase 4, whitespace (or the absence thereof) serves as more than preprocessing token separation. Whitespace can appear within a preprocessing token only as part of a header name or between the quotation characters in a character literal or string literal.

If the input stream has been parsed into preprocessing tokens up to a given character:

— If the next character begins a sequence of characters that can be the prefix and initial double quote of a raw string literal, such as `R"`, the next preprocessing token shall be a raw string literal. Between the initial and final double quote characters of the raw string, any transformations performed in phase 2 (line splicing) are reverted; this reversion shall apply before any *d-char*, *r-char*, or delimiting parenthesis is identified. The raw string literal is defined as the shortest sequence of characters that matches the raw-string pattern

  encoding-prefix$_{opt}$ `R` *raw-string*

— Otherwise, if the next three characters are `<::` and the subsequent character is neither `:` nor `>`, the `<` is treated as a preprocessing token by itself and not as the first character of the alternative token `<:`.

— Otherwise, the next preprocessing token is the longest sequence of characters that matches the syntax of a preprocessing token, even if that would cause further lexical analysis to fail, except that a *header-name* (5.8) is only formed

  — after the `include` or `import` preprocessing token in an `#include` (15.3) or `import` (15.5) directive, or

  — within a *has-include-expression*.

EXAMPLE 1

```
#define R "x"
const char* s = R"y";          // ill-formed raw string, not "x" "y"
```

The *import-keyword* is produced by processing an `import` directive (15.5), the *module-keyword* is produced by preprocessing a `module` directive (15.4), and the *export-keyword* is produced by preprocessing either of the previous two directives.

NOTE None has any observable spelling.

EXAMPLE 2 The program fragment `0xe+foo` is parsed as a preprocessing number token (one that is not a valid *integer-literal* or *floating-point-literal* token), even though a parse as three preprocessing tokens `0xe`, `+`, and `foo` can

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

produce a valid expression (for example, if `foo` is a macro defined as `1`). Similarly, the program fragment `1E1` is parsed as a preprocessing number (one that is a valid *floating-point-literal* token), whether or not `E` is a macro name.

EXAMPLE 3 The program fragment `x+++++y` is parsed as `x ++ ++ + y`, which, if `x` and `y` have integral types, violates a constraint on increment operators, even though the parse `x ++ + ++ y` can yield a correct expression.

5.5 Alternative tokens [lex.digraph]

Alternative token representations are provided for some operators and punctuators.[10]

In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling.[11] The set of alternative tokens is defined in Table 3.

Table 3 — Alternative tokens [tab:lex.digraph]

Alternative	Primary	Alternative	Primary	Alternative	Primary
`<%`	`{`	`and`	`&&`	`and_eq`	`&=`
`%>`	`}`	`bitor`	`|`	`or_eq`	`|=`
`<:`	`[`	`or`	`||`	`xor_eq`	`^=`
`:>`	`]`	`xor`	`^`	`not`	`!`
`%:`	`#`	`compl`	`~`	`not_eq`	`!=`
`%:%:`	`##`	`bitand`	`&`		

5.6 Tokens [lex.token]

token:
- *identifier*
- *keyword*
- *literal*
- *operator-or-punctuator*

There are five kinds of tokens: identifiers, keywords, literals,[12] operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collectively, "whitespace"), as described below, are ignored except as they serve to separate tokens.

NOTE Whitespace can separate otherwise adjacent identifiers, keywords, numeric literals, and alternative tokens containing alphabetic characters.

5.7 Comments [lex.comment]

The characters `/*` start a comment, which terminates with the characters `*/`. These comments do not nest. The characters `//` start a comment, which terminates immediately before the next new-line character. If there is a form-feed or a vertical-tab character in such a comment, only whitespace characters shall appear between it and the new-line that terminates the comment; no diagnostic is required.

NOTE The comment characters `//`, `/*`, and `*/` have no special meaning within a `//` comment and are treated just like other characters. Similarly, the comment characters `//` and `/*` have no special meaning within a `/*` comment.

5.8 Header names [lex.header]

header-name:
- `<` *h-char-sequence* `>`
- `"` *q-char-sequence* `"`

h-char-sequence:
- *h-char*
- *h-char-sequence h-char*

h-char:
- any member of the translation character set except new-line and U+003E GREATER-THAN SIGN

10) These include "digraphs" and additional reserved words. The term "digraph" (token consisting of two characters) is not perfectly descriptive, since one of the alternative *preprocessing-tokens* is `%:%:` and of course several primary tokens contain two characters. Nonetheless, those alternative tokens that aren't lexical keywords are colloquially known as "digraphs".

11) Thus the "stringized" values (15.6.3) of `[` and `<:` will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged.

12) Literals include strings and character and numeric literals.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

q-char-sequence:
  q-char
  q-char-sequence q-char

q-char:
  any member of the translation character set except new-line and U+0022 QUOTATION MARK

NOTE Header name preprocessing tokens only appear within a `#include` preprocessing directive, a `__has_include` preprocessing expression, or after certain occurrences of an `import` token (see 5.4).

The sequences in both forms of *header-name*s are mapped in an implementation-defined manner to headers or to external source file names as specified in 15.3.

The appearance of either of the characters `'` or `\` or of either of the character sequences `/*` or `//` in a *q-char-sequence* or an *h-char-sequence* is conditionally-supported with implementation-defined semantics, as is the appearance of the character `"` in an *h-char-sequence*.[13]

5.9 Preprocessing numbers [lex.ppnumber]

pp-number:
  digit
  `.` *digit*
  pp-number identifier-continue
  pp-number `'` *digit*
  pp-number `'` *nondigit*
  pp-number `e` *sign*
  pp-number `E` *sign*
  pp-number `p` *sign*
  pp-number `P` *sign*
  pp-number `.`

Preprocessing number tokens lexically include all *integer-literal* tokens (5.13.2) and all *floating-point-literal* tokens (5.13.4).

A preprocessing number does not have a type or a value; it acquires both after a successful conversion to an *integer-literal* token or a *floating-point-literal* token.

5.10 Identifiers [lex.name]

identifier:
  identifier-start
  identifier identifier-continue

identifier-start:
  nondigit
  an element of the translation character set with the Unicode property XID_Start

identifier-continue:
  digit
  nondigit
  an element of the translation character set with the Unicode property XID_Continue

nondigit: one of
  `a b c d e f g h i j k l m`
  `n o p q r s t u v w x y z`
  `A B C D E F G H I J K L M`
  `N O P Q R S T U V W X Y Z _`

digit: one of
  `0 1 2 3 4 5 6 7 8 9`

NOTE 1 The character properties XID_Start and XID_Continue are Derived Core Properties as described by UAX #44 of the Unicode Standard.[14]

The program is ill-formed if an *identifier* does not conform to Normalization Form C as specified in the Unicode Standard.

13) Thus, a sequence of characters that resembles an escape sequence can result in an error, be interpreted as the character corresponding to the escape sequence, or have a completely different meaning, depending on the implementation.

14) On systems in which linkers cannot accept extended characters, an encoding of the *universal-character-name* can be used in forming valid external identifiers. For example, some otherwise unused character or sequence of characters can be used to encode the `\u` in a *universal-character-name*. Extended characters can produce a long external identifier, but C++ does not place a translation limit on significant characters for external identifiers.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

NOTE 2 Identifiers are case-sensitive.

NOTE 3 Annex E compares the requirements of UAX #31 of the Unicode Standard with the C++ rules for identifiers.

NOTE 4 In translation phase 4, *identifier* also includes those *preprocessing-tokens* (5.4) differentiated as keywords (5.11) in the later translation phase 7 (5.6).

The identifiers in Table 4 have a special meaning when appearing in a certain context. When referred to in the grammar, these identifiers are used explicitly rather than using the *identifier* grammar production. Unless otherwise specified, any ambiguity as to whether a given *identifier* has a special meaning is resolved to interpret the token as a regular *identifier*.

Table 4 — Identifiers with special meaning [tab:lex.name.special]

`final`	`import`	`module`	`override`

In addition, some identifiers appearing as a *token* or *preprocessing-token* are reserved for use by C++ implementations and shall not be used otherwise; no diagnostic is required.

— Each identifier that contains a double underscore `__` or begins with an underscore followed by an uppercase letter is reserved to the implementation for any use.

— Each identifier that begins with an underscore is reserved to the implementation for use as a name in the global namespace.

5.11 Keywords [lex.key]

keyword:
- any identifier listed in Table 5
- *import-keyword*
- *module-keyword*
- *export-keyword*

The identifiers shown in Table 5 are reserved for use as keywords (that is, they are unconditionally treated as keywords in phase 7) except in an *attribute-token* (9.12.1).

NOTE The `register` keyword is unused but is reserved for future use.

Table 5 — Keywords [tab:lex.key]

`alignas`	`constinit`	`false`	`public`	`true`
`alignof`	`const_cast`	`float`	`register`	`try`
`asm`	`continue`	`for`	`reinterpret_cast`	`typedef`
`auto`	`co_await`	`friend`	`requires`	`typeid`
`bool`	`co_return`	`goto`	`return`	`typename`
`break`	`co_yield`	`if`	`short`	`union`
`case`	`decltype`	`inline`	`signed`	`unsigned`
`catch`	`default`	`int`	`sizeof`	`using`
`char`	`delete`	`long`	`static`	`virtual`
`char8_t`	`do`	`mutable`	`static_assert`	`void`
`char16_t`	`double`	`namespace`	`static_cast`	`volatile`
`char32_t`	`dynamic_cast`	`new`	`struct`	`wchar_t`
`class`	`else`	`noexcept`	`switch`	`while`
`concept`	`enum`	`nullptr`	`template`	
`const`	`explicit`	`operator`	`this`	
`consteval`	`export`	`private`	`thread_local`	
`constexpr`	`extern`	`protected`	`throw`	

Furthermore, the alternative representations shown in Table 6 for certain operators and punctuators (5.5) are reserved and shall not be used otherwise.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

Table 6 — Alternative representations [tab:lex.key.digraph]

`and`	`and_eq`	`bitand`	`bitor`	`compl`	`not`
`not_eq`	`or`	`or_eq`	`xor`	`xor_eq`	

5.12 Operators and punctuators [lex.operators]

The lexical representation of C++ programs includes a number of preprocessing tokens that are used in the syntax of the preprocessor or are converted into tokens for operators and punctuators:

preprocessing-op-or-punc:
  preprocessing-operator
  operator-or-punctuator

preprocessing-operator: one of
```
#      ##     %:     %:%:
```

operator-or-punctuator: one of
```
{      }      [      ]      (      )
<:     :>     <%     %>     ;      :      ...
?      ::     .      .*     ->     ->*    ~
!      +      -      *      /      %      ^      &      |
=      +=     -=     *=     /=     %=     ^=     &=     |=
==     !=     <      >      <=     >=     <=>    &&     ||
<<     >>     <<=    >>=    ++     --     ,
and    or     xor    not    bitand bitor  compl
and_eq or_eq  xor_eq not_eq
```

Each *operator-or-punctuator* is converted to a single token in translation phase 7 (5.2).

5.13 Literals [lex.literal]

5.13.1 Kinds of literals [lex.literal.kinds]

There are several kinds of literals.[15]

literal:
  integer-literal
  character-literal
  floating-point-literal
  string-literal
  boolean-literal
  pointer-literal
  user-defined-literal

NOTE When appearing as an *expression*, a literal has a type and a value category (7.5.2).

5.13.2 Integer literals [lex.icon]

integer-literal:
  binary-literal integer-suffix$_{opt}$
  octal-literal integer-suffix$_{opt}$
  decimal-literal integer-suffix$_{opt}$
  hexadecimal-literal integer-suffix$_{opt}$

binary-literal:
  `0b` *binary-digit*
  `0B` *binary-digit*
  binary-literal `'`$_{opt}$ *binary-digit*

octal-literal:
  `0`
  octal-literal `'`$_{opt}$ *octal-digit*

decimal-literal:
  nonzero-digit
  decimal-literal `'`$_{opt}$ *digit*

15) The term "literal" generally designates, in this document, those tokens that are called "constants" in C.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

hexadecimal-literal:
  hexadecimal-prefix hexadecimal-digit-sequence

binary-digit: one of
  `0 1`

octal-digit: one of
  `0 1 2 3 4 5 6 7`

nonzero-digit: one of
  `1 2 3 4 5 6 7 8 9`

hexadecimal-prefix: one of
  `0x 0X`

hexadecimal-digit-sequence:
  hexadecimal-digit
  hexadecimal-digit-sequence `'`$_{opt}$ *hexadecimal-digit*

hexadecimal-digit: one of
  `0 1 2 3 4 5 6 7 8 9`
  `a b c d e f`
  `A B C D E F`

integer-suffix:
  unsigned-suffix long-suffix$_{opt}$
  unsigned-suffix long-long-suffix$_{opt}$
  unsigned-suffix size-suffix$_{opt}$
  long-suffix unsigned-suffix$_{opt}$
  long-long-suffix unsigned-suffix$_{opt}$
  size-suffix unsigned-suffix$_{opt}$

unsigned-suffix: one of
  `u U`

long-suffix: one of
  `l L`

long-long-suffix: one of
  `ll LL`

size-suffix: one of
  `z Z`

In an *integer-literal*, the sequence of *binary-digit*s, *octal-digit*s, *digit*s, or *hexadecimal-digit*s is interpreted as a base N integer as shown in table Table 7; the lexically first digit of the sequence of digits is the most significant.

NOTE The prefix and any optional separating single quotes are ignored when determining the value.

Table 7 — Base of *integer-literal*s [tab:lex.icon.base]

Kind of *integer-literal*	base N
binary-literal	2
octal-literal	8
decimal-literal	10
hexadecimal-literal	16

The *hexadecimal-digit*s `a` through `f` and `A` through `F` have decimal values ten through fifteen.

EXAMPLE The number twelve can be written `12`, `014`, `0XC`, or `0b1100`. The *integer-literal*s `1048576`, `1'048'576`, `0X100000`, `0x10'0000`, and `0'004'000'000` all have the same value.

The type of an *integer-literal* is the first type in the list in Table 8 corresponding to its optional *integer-suffix* in which its value can be represented.

If an *integer-literal* cannot be represented by any type in its list and an extended integer type (6.8.2) can represent its value, it may have that extended integer type. If all of the types in the list for the *integer-literal* are signed, the extended integer type shall be signed. If all of the types in the list for the *integer-literal* are unsigned, the extended integer type shall be unsigned. If the list contains both signed and unsigned types,

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

the extended integer type may be signed or unsigned. A program is ill-formed if one of its translation units contains an *integer-literal* that cannot be represented by any of the allowed types.

Table 8 — Types of *integer-literal*s [tab:lex.icon.type]

integer-suffix	*decimal-literal*	*integer-literal* other than *decimal-literal*

none	`int`
	`long int`
	`long long int`
		`int`
		`unsigned int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`u` or `U`	`unsigned int`
	`unsigned long int`
	`unsigned long long int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long int`

`l` or `L`	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

Both `u` or `U` and `l` or `L`	`unsigned long int` `unsigned long long int`	`unsigned long int` `unsigned long long int`
`ll` or `LL`	`long long int`	`long long int` `unsigned long long int`
Both `u` or `U` and `ll` or `LL`	`unsigned long long int`	`unsigned long long int`
`z` or `Z`	the signed integer type corresponding to `std::size_t` (17.2.4)	the signed integer type corresponding to `std::size_t` `std::size_t`
Both `u` or `U` and `z` or `Z`	`std::size_t`	`std::size_t`

5.13.3 Character literals [lex.ccon]

character-literal:
  encoding-prefix$_{opt}$ ' *c-char-sequence* '

encoding-prefix: one of
  `u8` `u` `U` `L`

c-char-sequence:
  c-char
  c-char-sequence c-char

c-char:
  basic-c-char
  escape-sequence
  universal-character-name

basic-c-char:
  any member of the translation character set except the U+0027 APOSTROPHE, U+005C REVERSE SOLIDUS, or new-line character

escape-sequence:
  simple-escape-sequence
  numeric-escape-sequence
  conditional-escape-sequence

simple-escape-sequence:
  \ *simple-escape-sequence-char*

simple-escape-sequence-char: one of
  `' " ? \ a b f n r t v`

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

numeric-escape-sequence:
  octal-escape-sequence
  hexadecimal-escape-sequence

simple-octal-digit-sequence:
  octal-digit
  simple-octal-digit-sequence octal-digit

octal-escape-sequence:
  \ *octal-digit*
  \ *octal-digit octal-digit*
  \ *octal-digit octal-digit octal-digit*
  \o{ *simple-octal-digit-sequence* }

hexadecimal-escape-sequence:
  \x *simple-hexadecimal-digit-sequence*
  \x{ *simple-hexadecimal-digit-sequence* }

conditional-escape-sequence:
  \ *conditional-escape-sequence-char*

conditional-escape-sequence-char:
  any member of the basic character set that is not an *octal-digit*, a *simple-escape-sequence-char*, or the characters `N`, `o`, `u`, `U`, or `x`

A *non-encodable character literal* is a *character-literal* whose *c-char-sequence* consists of a single *c-char* that is not a *numeric-escape-sequence* and that specifies a character that either lacks representation in the literal's associated character encoding or that cannot be encoded as a single code unit. A *multicharacter literal* is a *character-literal* whose *c-char-sequence* consists of more than one *c-char*. The *encoding-prefix* of a non-encodable character literal or a multicharacter literal shall be absent. Such *character-literal*s are conditionally-supported.

The kind of a *character-literal*, its type, and its associated character encoding (5.3) are determined by its *encoding-prefix* and its *c-char-sequence* as defined by Table 9. The special cases for non-encodable character literals and multicharacter literals take precedence over the base kind.

NOTE 1 The associated character encoding for ordinary character literals determines encodability, but does not determine the value of non-encodable ordinary character literals or ordinary multicharacter literals. The examples in Table 9 for non-encodable ordinary character literals assume that the specified character lacks representation in the ordinary literal encoding or that encoding the character would require more than one code unit.

Table 9 — Character literals [tab:lex.ccon.literal]

Encoding prefix	Kind	Type	Associated character encoding	Example
none	*ordinary character literal*	`char`	ordinary literal encoding	`'v'`
	non-encodable ordinary character literal	`int`		`'\U0001F525'`
	ordinary multicharacter literal	`int`		`'abcd'`
`L`	*wide character literal*	`wchar_t`	wide literal encoding	`L'w'`
`u8`	*UTF-8 character literal*	`char8_t`	UTF-8	`u8'x'`
`u`	*UTF-16 character literal*	`char16_t`	UTF-16	`u'y'`
`U`	*UTF-32 character literal*	`char32_t`	UTF-32	`U'z'`

In translation phase 4, the value of a *character-literal* is determined using the range of representable values of the *character-literal*'s type in translation phase 7. A non-encodable character literal or a multicharacter literal has an implementation-defined value. The value of any other kind of *character-literal* is determined as follows:

— A *character-literal* with a *c-char-sequence* consisting of a single *basic-c-char*, *simple-escape-sequence*, or *universal-character-name* is the code unit value of the specified character as encoded in the literal's associated character encoding.

NOTE 2 If the specified character lacks representation in the literal's associated character encoding or if it cannot be encoded as a single code unit, then the literal is a non-encodable character literal.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

— A *character-literal* with a *c-char-sequence* consisting of a single *numeric-escape-sequence* has a value as follows:

  — Let v be the integer value represented by the octal number comprising the sequence of *octal-digit*s in an *octal-escape-sequence* or by the hexadecimal number comprising the sequence of *hexadecimal-digit*s in a *hexadecimal-escape-sequence*.

  — If v does not exceed the range of representable values of the *character-literal*'s type, then the value is v.

  — Otherwise, if the *character-literal*'s *encoding-prefix* is absent or `L`, and v does not exceed the range of representable values of the corresponding unsigned type for the underlying type of the *character-literal*'s type, then the value is the unique value of the *character-literal*'s type `T` that is congruent to v modulo 2^N, where N is the width of `T`.

  — Otherwise, the *character-literal* is ill-formed.

— A *character-literal* with a *c-char-sequence* consisting of a single *conditional-escape-sequence* is conditionally-supported and has an implementation-defined value.

The character specified by a *simple-escape-sequence* is specified in Table 10.

NOTE 3 Using an escape sequence for a question mark is supported for compatibility with ISO/IEC 14882:2014 and C.

Table 10 — Simple escape sequences [tab:lex.ccon.esc]

character		*simple-escape-sequence*
U+000A	LINE FEED	`\n`
U+0009	CHARACTER TABULATION	`\t`
U+000B	LINE TABULATION	`\v`
U+0008	BACKSPACE	`\b`
U+000D	CARRIAGE RETURN	`\r`
U+000C	FORM FEED	`\f`
U+0007	ALERT	`\a`
U+005C	REVERSE SOLIDUS	`\\`
U+003F	QUESTION MARK	`\?`
U+0027	APOSTROPHE	`\'`
U+0022	QUOTATION MARK	`\"`

5.13.4 Floating-point literals [lex.fcon]

floating-point-literal:
  decimal-floating-point-literal
  hexadecimal-floating-point-literal

decimal-floating-point-literal:
  fractional-constant exponent-part$_{opt}$ *floating-point-suffix*$_{opt}$
  digit-sequence exponent-part floating-point-suffix$_{opt}$

hexadecimal-floating-point-literal:
  hexadecimal-prefix hexadecimal-fractional-constant binary-exponent-part floating-point-suffix$_{opt}$
  hexadecimal-prefix hexadecimal-digit-sequence binary-exponent-part floating-point-suffix$_{opt}$

fractional-constant:
  digit-sequence$_{opt}$ `.` *digit-sequence*
  digit-sequence `.`

hexadecimal-fractional-constant:
  hexadecimal-digit-sequence$_{opt}$ `.` *hexadecimal-digit-sequence*
  hexadecimal-digit-sequence `.`

exponent-part:
  `e` *sign*$_{opt}$ *digit-sequence*
  `E` *sign*$_{opt}$ *digit-sequence*

binary-exponent-part:
  `p` *sign*$_{opt}$ *digit-sequence*
  `P` *sign*$_{opt}$ *digit-sequence*

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

sign: one of
`+` `-`

digit-sequence:
digit
digit-sequence `'`$_{opt}$ *digit*

floating-point-suffix: one of
`f l f16 f32 f64 f128 bf16 F L F16 F32 F64 F128 BF16`

The type of a *floating-point-literal* (6.8.2, 6.8.3) is determined by its *floating-point-suffix* as specified in Table 11.

NOTE 1 The floating-point suffixes `f16`, `f32`, `f64`, `f128`, `bf16`, `F16`, `F32`, `F64`, `F128`, and `BF16` are conditionally-supported. See 6.8.3.

Table 11 — Types of *floating-point-literal*s [tab:lex.fcon.type]

floating-point-suffix	type
none	`double`
`f` or `F`	`float`
`l` or `L`	`long double`
`f16` or `F16`	`std::float16_t`
`f32` or `F32`	`std::float32_t`
`f64` or `F64`	`std::float64_t`
`f128` or `F128`	`std::float128_t`
`bf16` or `BF16`	`std::bfloat16_t`

The *significand* of a *floating-point-literal* is the *fractional-constant* or *digit-sequence* of a *decimal-floating-point-literal* or the *hexadecimal-fractional-constant* or *hexadecimal-digit-sequence* of a *hexadecimal-floating-point-literal*. In the significand, the sequence of *digit*s or *hexadecimal-digit*s and optional period are interpreted as a base N real number s, where N is 10 for a *decimal-floating-point-literal* and 16 for a *hexadecimal-floating-point-literal*.

NOTE 2 Any optional separating single quotes are ignored when determining the value.

If an *exponent-part* or *binary-exponent-part* is present, the exponent e of the *floating-point-literal* is the result of interpreting the sequence of an optional *sign* and the *digit*s as a base 10 integer. Otherwise, the exponent e is 0. The scaled value of the literal is $s \times 10^e$ for a *decimal-floating-point-literal* and $s \times 2^e$ for a *hexadecimal-floating-point-literal*.

EXAMPLE The *floating-point-literal*s `1.602'176'565e-19` and `1.602176565e-19` have the same value. The *floating-point-literal*s `49.625` and `0xC.68p+2` have the same value.

If the scaled value is not in the range of representable values for its type, the program is ill-formed. Otherwise, the value of a *floating-point-literal* is the scaled value if representable, else the larger or smaller representable value nearest the scaled value, chosen in an implementation-defined manner.

5.13.5 String literals [lex.string]

string-literal:
encoding-prefix$_{opt}$ `"` *s-char-sequence*$_{opt}$ `"`
encoding-prefix$_{opt}$ `R` *raw-string*

s-char-sequence:
s-char
s-char-sequence s-char

s-char:
basic-s-char
escape-sequence
universal-character-name

basic-s-char:
any member of the translation character set except the U+0022 QUOTATION MARK, U+005C REVERSE SOLIDUS, or new-line character

raw-string:
`"` *d-char-sequence*$_{opt}$ `(` *r-char-sequence*$_{opt}$ `)` *d-char-sequence*$_{opt}$ `"`

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

r-char-sequence:
r-char
r-char-sequence r-char

r-char:
any member of the translation character set, except a U+0029 RIGHT PARENTHESIS followed by the initial *d-char-sequence* (which may be empty) followed by a U+0022 QUOTATION MARK

d-char-sequence:
d-char
d-char-sequence d-char

d-char:
any member of the basic character set except:
U+0020 SPACE, U+0028 LEFT PARENTHESIS, U+0029 RIGHT PARENTHESIS, U+005C REVERSE SOLIDUS, U+0009 CHARACTER TABULATION, U+000B LINE TABULATION, U+000C FORM FEED, and new-line

The kind of a *string-literal*, its type, and its associated character encoding (5.3) are determined by its encoding prefix and sequence of *s-char*s or *r-char*s as defined by Table 12 where n is the number of encoded code units as described below.

Table 12 — String literals [tab:lex.string.literal]

Encoding prefix	Kind	Type	Associated character encoding	Examples
none	*ordinary string literal*	array of n `const char`	ordinary literal encoding	`"ordinary string"` `R"(ordinary raw string)"`
`L`	*wide string literal*	array of n `const wchar_t`	wide literal encoding	`L"wide string"` `LR"w(wide raw string)w"`
`u8`	*UTF-8 string literal*	array of n `const char8_t`	UTF-8	`u8"UTF-8 string"` `u8R"x(UTF-8 raw string)x"`
`u`	*UTF-16 string literal*	array of n `const char16_t`	UTF-16	`u"UTF-16 string"` `uR"y(UTF-16 raw string)y"`
`U`	*UTF-32 string literal*	array of n `const char32_t`	UTF-32	`U"UTF-32 string"` `UR"z(UTF-32 raw string)z"`

A *string-literal* that has an `R` in the prefix is a *raw string literal*. The *d-char-sequence* serves as a delimiter. The terminating *d-char-sequence* of a *raw-string* is the same sequence of characters as the initial *d-char-sequence*. A *d-char-sequence* shall consist of at most 16 characters.

NOTE 1 The characters `'('` and `')'` can appear in a *raw-string*. Thus, `R"delimiter((a|b))delimiter"` is equivalent to `"(a|b)"`.

NOTE 2 A source-file new-line in a raw string literal results in a new-line in the resulting execution string literal. Assuming no whitespace at the beginning of lines in the following example, the assert will succeed:

```
const char* p = R"(a\
b
c)";
assert(std::strcmp(p, "a\\\nb\nc") == 0);
```

EXAMPLE 1 The raw string

```
R"a(
)\
a"
)a"
```

is equivalent to `"\n)\\\na\"\n"`. The raw string

```
R"(x = "\"y\"")"
```

is equivalent to `"x = \"\\\"y\\\"\""`.

Ordinary string literals and UTF-8 string literals are also referred to as narrow string literals.

The common *encoding-prefix* for a sequence of adjacent *string-literal*s is determined pairwise as follows. If two *string-literal*s have the same *encoding-prefix*, the common *encoding-prefix* is that *encoding-prefix*. If one

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

string-literal has no *encoding-prefix*, the common *encoding-prefix* is that of the other *string-literal*. Any other combinations are ill-formed.

NOTE 3 A *string-literal*'s rawness has no effect on the determination of the common *encoding-prefix*.

In translation phase 6 (5.2), adjacent *string-literal*s are concatenated. The lexical structure and grouping of the contents of the individual *string-literal*s is retained.

EXAMPLE 2

```
"\xA" "B"
```

represents the code unit `'\xA'` and the character `'B'` after concatenation (and not the single code unit `'\xAB'`). Similarly,

```
R"(\u00)" "41"
```

represents six characters, starting with a backslash and ending with the digit `1` (and not the single character `'A'` specified by a *universal-character-name*).

Table 13 has some examples of valid concatenations.

Table 13 — String literal concatenations [tab:lex.string.concat]

Source		Means	Source		Means	Source		Means
`u"a"`	`u"b"`	`u"ab"`	`U"a"`	`U"b"`	`U"ab"`	`L"a"`	`L"b"`	`L"ab"`
`u"a"`	`"b"`	`u"ab"`	`U"a"`	`"b"`	`U"ab"`	`L"a"`	`"b"`	`L"ab"`
`"a"`	`u"b"`	`u"ab"`	`"a"`	`U"b"`	`U"ab"`	`"a"`	`L"b"`	`L"ab"`

Evaluating a *string-literal* results in a string literal object with static storage duration (6.7.5). Whether all *string-literal*s are distinct (that is, are stored in nonoverlapping objects) and whether successive evaluations of a *string-literal* yield the same or a different object is unspecified.

NOTE 4 The effect of attempting to modify a string literal object is undefined.

String literal objects are initialized with the sequence of code unit values corresponding to the *string-literal*'s sequence of *s-char*s (originally from non-raw string literals) and *r-char*s (originally from raw string literals), plus a terminating U+0000 NULL character, in order as follows:

— The sequence of characters denoted by each contiguous sequence of *basic-s-char*s, *r-char*s, *simple-escape-sequence*s (5.13.3), and *universal-character-name*s (5.3) is encoded to a code unit sequence using the *string-literal*'s associated character encoding. If a character lacks representation in the associated character encoding, then the *string-literal* is conditionally-supported and an implementation-defined code unit sequence is encoded.

NOTE 5 No character lacks representation in any Unicode encoding form.

When encoding a stateful character encoding, implementations should encode the first such sequence beginning with the initial encoding state and encode subsequent sequences beginning with the final encoding state of the prior sequence.

NOTE 6 The encoded code unit sequence can differ from the sequence of code units that would be obtained by encoding each character independently.

— Each *numeric-escape-sequence* (5.13.3) contributes a single code unit with a value as follows:

— Let v be the integer value represented by the octal number comprising the sequence of *octal-digit*s in an *octal-escape-sequence* or by the hexadecimal number comprising the sequence of *hexadecimal-digit*s in a *hexadecimal-escape-sequence*.

— If v does not exceed the range of representable values of the *string-literal*'s array element type, then the value is v.

— Otherwise, if the *string-literal*'s *encoding-prefix* is absent or `L`, and v does not exceed the range of representable values of the corresponding unsigned type for the underlying type of the *string-literal*'s array element type, then the value is the unique value of the *string-literal*'s array element type `T` that is congruent to v modulo 2^N, where N is the width of `T`.

— Otherwise, the *string-literal* is ill-formed.

When encoding a stateful character encoding, these sequences should have no effect on encoding state.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

— Each *conditional-escape-sequence* (5.13.3) contributes an implementation-defined code unit sequence. When encoding a stateful character encoding, it is implementation-defined what effect these sequences have on encoding state.

5.13.6 Boolean literals [lex.bool]

boolean-literal:
`false`
`true`

The Boolean literals are the keywords `false` and `true`. Such literals have type `bool`.

5.13.7 Pointer literals [lex.nullptr]

pointer-literal:
`nullptr`

The pointer literal is the keyword `nullptr`. It has type `std::nullptr_t`.

NOTE `std::nullptr_t` is a distinct type that is neither a pointer type nor a pointer-to-member type; rather, a prvalue of this type is a null pointer constant and can be converted to a null pointer value or null member pointer value. See 7.3.12 and 7.3.13.

5.13.8 User-defined literals [lex.ext]

user-defined-literal:
user-defined-integer-literal
user-defined-floating-point-literal
user-defined-string-literal
user-defined-character-literal

user-defined-integer-literal:
decimal-literal ud-suffix
octal-literal ud-suffix
hexadecimal-literal ud-suffix
binary-literal ud-suffix

user-defined-floating-point-literal:
fractional-constant exponent-part$_{opt}$ *ud-suffix*
digit-sequence exponent-part ud-suffix
hexadecimal-prefix hexadecimal-fractional-constant binary-exponent-part ud-suffix
hexadecimal-prefix hexadecimal-digit-sequence binary-exponent-part ud-suffix

user-defined-string-literal:
string-literal ud-suffix

user-defined-character-literal:
character-literal ud-suffix

ud-suffix:
identifier

If a token matches both *user-defined-literal* and another *literal* kind, it is treated as the latter.

EXAMPLE 1 `123_km` is a *user-defined-literal*, but `12LL` is an *integer-literal*.

The syntactic non-terminal preceding the *ud-suffix* in a *user-defined-literal* is taken to be the longest sequence of characters that matches the syntax of that non-terminal.

A *user-defined-literal* is treated as a call to a literal operator or literal operator template (12.6). To determine the form of this call for a given *user-defined-literal* L with *ud-suffix* X, first let S be the set of declarations found by unqualified lookup for the *literal-operator-id* whose literal suffix identifier is X (6.5.3). S shall not be empty.

If L is a *user-defined-integer-literal*, let n be the literal without its *ud-suffix*. If S contains a literal operator with parameter type `unsigned long long`, the literal L is treated as a call of the form

```
operator ""X(nULL)
```

Otherwise, S shall contain a raw literal operator or a numeric literal operator template (12.6) but not both. If S contains a raw literal operator, the literal L is treated as a call of the form

```
operator ""X("n")
```

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

Otherwise (S contains a numeric literal operator template), L is treated as a call of the form

```
operator ""X<'c1', 'c2', ... 'ck'>()
```

where n is the source character sequence $c_1c_2...c_k$.

NOTE 1 The sequence $c_1c_2...c_k$ can only contain characters from the basic character set.

If L is a *user-defined-floating-point-literal*, let f be the literal without its *ud-suffix*. If S contains a literal operator with parameter type `long double`, the literal L is treated as a call of the form

```
operator ""X(fL)
```

Otherwise, S shall contain a raw literal operator or a numeric literal operator template (12.6) but not both. If S contains a raw literal operator, the *literal* L is treated as a call of the form

```
operator ""X("f")
```

Otherwise (S contains a numeric literal operator template), L is treated as a call of the form

```
operator ""X<'c1', 'c2', ... 'ck'>()
```

where f is the source character sequence $c_1c_2...c_k$.

NOTE 2 The sequence $c_1c_2...c_k$ can only contain characters from the basic character set.

If L is a *user-defined-string-literal*, let *str* be the literal without its *ud-suffix* and let *len* be the number of code units in *str* (i.e., its length excluding the terminating null character). If S contains a literal operator template with a non-type template parameter for which *str* is a well-formed *template-argument*, the literal L is treated as a call of the form

```
operator ""X<str>()
```

Otherwise, the literal L is treated as a call of the form

```
operator ""X(str, len)
```

If L is a *user-defined-character-literal*, let *ch* be the literal without its *ud-suffix*. S shall contain a literal operator (12.6) whose only parameter has the type of *ch* and the literal L is treated as a call of the form

```
operator ""X(ch)
```

EXAMPLE 2

```
long double operator ""_w(long double);
std::string operator ""_w(const char16_t*, std::size_t);
unsigned operator ""_w(const char*);
int main() {
  1.2_w;            // calls operator ""_w(1.2L)
  u"one"_w;         // calls operator ""_w(u"one", 3)
  12_w;             // calls operator ""_w("12")
  "two"_w;          // error: no applicable literal operator
}
```

In translation phase 6 (5.2), adjacent *string-literal*s are concatenated and *user-defined-string-literal*s are considered *string-literal*s for that purpose. During concatenation, *ud-suffix*es are removed and ignored and the concatenation process occurs as described in 5.13.5. At the end of phase 6, if a *string-literal* is the result of a concatenation involving at least one *user-defined-string-literal*, all the participating *user-defined-string-literal*s shall have the same *ud-suffix* and that suffix is applied to the result of the concatenation.

EXAMPLE 3

```
int main() {
  L"A" "B" "C"_x;    // OK, same as L"ABC"_x
  "P"_x "Q" "R"_y;   // error: two different ud-suffixes
}
```

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

6 Basics [basic]

6.1 Preamble [basic.pre]

NOTE 1 This Clause presents the basic concepts of the C++ language. It explains the difference between an object and a name and how they relate to the value categories for expressions. It introduces the concepts of a declaration and a definition and presents C++'s notion of type, scope, linkage, and storage duration. The mechanisms for starting and terminating a program are discussed. Finally, this Clause presents the fundamental types of the language and lists the ways of constructing compound types from these.

NOTE 2 This Clause does not cover concepts that affect only a single part of the language. Such concepts are discussed in Clause 7 through Clause 15.

An *entity* is a value, object, reference, structured binding, function, enumerator, type, class member, bit-field, template, template specialization, namespace, or pack.

A *name* is an *identifier* (5.10), *operator-function-id* (12.4), *literal-operator-id* (12.6), or *conversion-function-id* (11.4.8.3).

Every name is introduced by a *declaration*, which is a

— *name-declaration*, *block-declaration*, or *member-declaration* (9.1, 11.4),

— *init-declarator* (9.3),

— *identifier* in a structured binding declaration (9.6),

— *init-capture* (7.5.6.3),

— *condition* with a *declarator* (8.1),

— *member-declarator* (11.4),

— *using-declarator* (9.9),

— *parameter-declaration* (9.3.4.6),

— *type-parameter* (13.2),

— *elaborated-type-specifier* that introduces a name (9.2.9.4),

— *class-specifier* (11.1),

— *enum-specifier* or *enumerator-definition* (9.7.1),

— *exception-declaration* (14.1), or

— implicit declaration of an injected-class-name (11.1).

NOTE 3 The interpretation of a *for-range-declaration* produces one or more of the above (8.6.5).

An entity E is denoted by the name (if any) that is introduced by a declaration of E or by a *typedef-name* introduced by a declaration specifying E.

A *variable* is introduced by the declaration of a reference other than a non-static data member or of an object. The variable's name, if any, denotes the reference or object.

A *local entity* is a variable with automatic storage duration (6.7.5.4), a structured binding (9.6) whose corresponding variable is such an entity, or the `*this` object (7.5.3).

Some names denote types or templates. In general, whenever a name is encountered it is necessary to determine whether that name denotes one of these entities before continuing to parse the program that contains it. The process that determines this is called *name lookup* (6.5).

Two names are *the same* if

— they are *identifier*s composed of the same character sequence, or

— they are *operator-function-id*s formed with the same operator, or

— they are *conversion-function-id*s formed with equivalent (13.7.7.2) types, or

— they are *literal-operator-id*s (12.6) formed with the same literal suffix identifier.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

A name used in more than one translation unit can potentially refer to the same entity in these translation units depending on the linkage (6.6) of the name specified in each translation unit.

6.2 Declarations and definitions [basic.def]

A declaration (Clause 9) may (re)introduce one or more names and/or entities into a translation unit. If so, the declaration specifies the interpretation and semantic properties of these names. A declaration of an entity or *typedef-name* X is a redeclaration of X if another declaration of X is reachable from it (10.7). A declaration may also have effects including:

— a static assertion (9.1),

— controlling template instantiation (13.9.3),

— guiding template argument deduction for constructors (13.7.2.3),

— use of attributes (9.12), and

— nothing (in the case of an *empty-declaration*).

Each entity declared by a *declaration* is also *defined* by that declaration unless:

— it declares a function without specifying the function's body (9.5),

— it contains the `extern` specifier (9.2.2) or a *linkage-specification*[16] (9.11) and neither an *initializer* nor a *function-body*,

— it declares a non-inline static data member in a class definition (11.4, 11.4.9),

— it declares a static data member outside a class definition and the variable was defined within the class with the `constexpr` specifier (this usage is deprecated; see D.6),

— it is an *elaborated-type-specifier* (11.3),

— it is an *opaque-enum-declaration* (9.7.1),

— it is a *template-parameter* (13.2),

— it is a *parameter-declaration* (9.3.4.6) in a function declarator that is not the *declarator* of a *function-definition*,

— it is a `typedef` declaration (9.2.4),

— it is an *alias-declaration* (9.2.4),

— it is a *using-declaration* (9.9),

— it is a *deduction-guide* (13.7.2.3),

— it is a *static_assert-declaration* (9.1),

— it is an *attribute-declaration* (9.1),

— it is an *empty-declaration* (9.1),

— it is a *using-directive* (9.8.4),

— it is a *using-enum-declaration* (9.7.2),

— it is a *template-declaration* (13.1) whose *template-head* is not followed by either a *concept-definition* or a *declaration* that defines a function, a class, a variable, or a static data member.

— it is an explicit instantiation declaration (13.9.3), or

— it is an explicit specialization (13.9.4) whose *declaration* is not a definition.

A declaration is said to be a *definition* of each entity that it defines.

EXAMPLE 1 All but one of the following are definitions:

```
int a;                          // defines a
extern const int c = 1;         // defines c
int f(int x) { return x+a; }    // defines f and defines x
struct S { int a; int b; };     // defines S, S::a, and S::b
struct X {                      // defines X
  int x;                        // defines non-static data member x
```

16) Appearing inside the brace-enclosed *declaration-seq* in a *linkage-specification* does not affect whether a declaration is a definition.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

```
  static int y;                 // declares static data member y
  X(): x(0) { }                 // defines a constructor of X
};
int X::y = 1;                   // defines X::y
enum { up, down };              // defines up and down
namespace N { int d; }          // defines N and N::d
namespace N1 = N;               // defines N1
X anX;                          // defines anX
```

whereas these are just declarations:

```
extern int a;                   // declares a
extern const int c;             // declares c
int f(int);                     // declares f
struct S;                       // declares S
typedef int Int;                // declares Int
extern X anotherX;              // declares anotherX
using N::d;                     // declares d
```

NOTE 1 In some circumstances, C++ implementations implicitly define the default constructor (11.4.5.2), copy constructor, move constructor (11.4.5.3), copy assignment operator, move assignment operator (11.4.6), or destructor (11.4.7) member functions.

EXAMPLE 2 Given

```
#include <string>

struct C {
  std::string s;                // std::string is the standard library class (23.4)
};

int main() {
  C a;
  C b = a;
  b = a;
}
```

the implementation will implicitly define functions to make the definition of `C` equivalent to

```
struct C {
  std::string s;
  C() : s() { }
  C(const C& x): s(x.s) { }
  C(C&& x): s(static_cast<std::string&&>(x.s)) { }
      //  : s(std::move(x.s)) { }
  C& operator=(const C& x) { s = x.s; return *this; }
  C& operator=(C&& x) { s = static_cast<std::string&&>(x.s); return *this; }
      //              { s = std::move(x.s); return *this; }
  ~C() { }
};
```

NOTE 2 A class name can also be implicitly declared by an *elaborated-type-specifier* (9.2.9.4).

In the definition of an object, the type of that object shall not be an incomplete type (6.8.1), an abstract class type (11.7.4), or a (possibly multidimensional) array thereof.

6.3 One-definition rule [basic.def.odr]

Each of the following is termed a *definable item*:

— a class type (Clause 11),

— an enumeration type (9.7.1),

— a function (9.3.4.6),

— a variable (6.1),

— a templated entity (13.1),

— a default argument for a parameter (for a function in a given scope) (9.3.4.7), or

— a default template argument (13.2).

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

No translation unit shall contain more than one definition of any definable item.

An expression or conversion is *potentially evaluated* unless it is an unevaluated operand (7.2.3), a subexpression thereof, or a conversion in an initialization or conversion sequence in such a context. The set of *potential results* of an expression E is defined as follows:

— If E is an *id-expression* (7.5.5), the set contains only E.

— If E is a subscripting operation (7.6.1.2) with an array operand, the set contains the potential results of that operand.

— If E is a class member access expression (7.6.1.5) of the form E_1 `.` `template`$_{opt}$ E_2 naming a non-static data member, the set contains the potential results of E_1.

— If E is a class member access expression naming a static data member, the set contains the *id-expression* designating the data member.

— If E is a pointer-to-member expression (7.6.4) of the form E_1 `.*` E_2, the set contains the potential results of E_1.

— If E has the form `(`E_1`)`, the set contains the potential results of E_1.

— If E is a glvalue conditional expression (7.6.16), the set is the union of the sets of potential results of the second and third operands.

— If E is a comma expression (7.6.20), the set contains the potential results of the right operand.

— Otherwise, the set is empty.

NOTE 1 This set is a (possibly-empty) set of *id-expressions*, each of which is either E or a subexpression of E.

EXAMPLE 1 In the following example, the set of potential results of the initializer of `n` contains the first `S::x` subexpression, but not the second `S::x` subexpression.

```
struct S { static const int x = 0; };
const int &f(const int &r);
int n = b ? (1, S::x)          // S::x is not odr-used here
          : f(S::x);           // S::x is odr-used here, so a definition is required
```

A function is *named by* an expression or conversion as follows:

— A function is named by an expression or conversion if it is the selected member of an overload set (6.5, 12.2, 12.3) in an overload resolution performed as part of forming that expression or conversion, unless it is a pure virtual function and either the expression is not an *id-expression* naming the function with an explicitly qualified name or the expression forms a pointer to member (7.6.2.2).

NOTE 2 This covers taking the address of functions (7.3.4, 7.6.2.2), calls to named functions (7.6.1.3), operator overloading (Clause 12), user-defined conversions (11.4.8.3), allocation functions for *new-expressions* (7.6.2.8), as well as non-default initialization (9.4). A constructor selected to copy or move an object of class type is considered to be named by an expression or conversion even if the call is actually elided by the implementation (11.9.6).

— A deallocation function for a class is named by a *new-expression* if it is the single matching deallocation function for the allocation function selected by overload resolution, as specified in 7.6.2.8.

— A deallocation function for a class is named by a *delete-expression* if it is the selected usual deallocation function as specified in 7.6.2.9 and 11.4.11.

A variable is named by an expression if the expression is an *id-expression* that denotes it. A variable `x` that is named by a potentially-evaluated expression E is *odr-used* by E unless

— `x` is a reference that is usable in constant expressions (7.7), or

— `x` is a variable of non-reference type that is usable in constant expressions and has no mutable subobjects, and E is an element of the set of potential results of an expression of non-volatile-qualified non-class type to which the lvalue-to-rvalue conversion (7.3.2) is applied, or

— `x` is a variable of non-reference type, and E is an element of the set of potential results of a discarded-value expression (7.2.3) to which the lvalue-to-rvalue conversion is not applied.

A structured binding is odr-used if it appears as a potentially-evaluated expression.

`*this` is odr-used if `this` appears as a potentially-evaluated expression (including as the result of the implicit transformation in the body of a non-static member function (11.4.3)).

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024

A virtual member function is odr-used if it is not pure. A function is odr-used if it is named by a potentially-evaluated expression or conversion. A non-placement allocation or deallocation function for a class is odr-used by the definition of a constructor of that class. A non-placement deallocation function for a class is odr-used by the definition of the destructor of that class, or by being selected by the lookup at the point of definition of a virtual destructor (11.4.7).[17]

An assignment operator function in a class is odr-used by an implicitly-defined copy-assignment or move-assignment function for another class as specified in 11.4.6. A constructor for a class is odr-used as specified in 9.4. A destructor for a class is odr-used if it is potentially invoked (11.4.7).

A local entity (6.1) is *odr-usable* in a scope (6.4.1) if

— either the local entity is not `*this`, or an enclosing class or non-lambda function parameter scope exists and, if the innermost such scope is a function parameter scope, it corresponds to a non-static member function, and

— for each intervening scope (6.4.1) between the point at which the entity is introduced and the scope (where `*this` is considered to be introduced within the innermost enclosing class or non-lambda function definition scope), either:

  — the intervening scope is a block scope, or

  — the intervening scope is the function parameter scope of a *lambda-expression* that has a *simple-capture* naming the entity or has a *capture-default*, and the block scope of the *lambda-expression* is also an intervening scope.

If a local entity is odr-used in a scope in which it is not odr-usable, the program is ill-formed.

EXAMPLE 2

```
void f(int n) {
  [] { n = 1; };                  // error: n is not odr-usable due to intervening lambda-expression
  struct A {
    void f() { n = 2; }           // error: n is not odr-usable due to intervening function definition scope
  };
  void g(int = n);                // error: n is not odr-usable due to intervening function parameter scope
  [=](int k = n) {};              // error: n is not odr-usable due to being
                                  // outside the block scope of the lambda-expression
  [&] { [n]{ return n; }; };      // OK
}
```

Every program shall contain at least one definition of every function or variable that is odr-used in that program outside of a discarded statement (8.5.2); no diagnostic required. The definition can appear explicitly in the program, it can be found in the standard or a user-defined library, or (when appropriate) it is implicitly defined (see 11.4.5.2, 11.4.5.3, 11.4.7, and 11.4.6).

EXAMPLE 3

```
auto f() {
  struct A {};
  return A{};
}
decltype(f()) g();
auto x = g();
```

A program containing this translation unit is ill-formed because `g` is odr-used but not defined, and cannot be defined in any other translation unit because the local class `A` cannot be named outside this translation unit.

A *definition domain* is a *private-module-fragment* or the portion of a translation unit excluding its *private-module-fragment* (if any). A definition of an inline function or variable shall be reachable from the end of every definition domain in which it is odr-used outside of a discarded statement.

A definition of a class shall be reachable in every context in which the class is used in a way that requires the class type to be complete.

EXAMPLE 4 The following complete translation unit is well-formed, even though it never defines `X`:

```
struct X;                         // declare X as a struct type
struct X* x1;                     // use X in pointer formation
```

17) An implementation is not required to call allocation and deallocation functions from constructors or destructors; however, this is a permissible implementation technique.

©ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2024